Kubernetes Handbook

Jimmy Song

目录

前言

云原生

概念与原理

用户指南

最佳实践

领域应用

开发指南

附录

Kubernetes Handbook

Kubernetes是Google基于Borg开源的容器编排调度引擎，作为CNCF（Cloud Native Computing Foundation）最重要的组件之一，它的目标不仅仅是一个编排系统，而是提供一个规范，可以让你来描述集群的架构，定义服务的最终状态，kubernetes可以帮你将系统自动地达到和维持在这个状态。Kubernetes作为云原生应用的基石，相当于一个云操作系统，其重要性不言而喻。

本书记录了本人从零开始学习和使用Kubernetes的心路历程，着重于经验分享和总结，同时也会有相关的概念解析，希望能够帮助大家少踩坑，少走弯路，还会指引大家关于关注kubernetes生态周边，如微服务构建、DevOps、大数据应用、Service Mesh、Cloud Native等领域。

本书的主题不仅限于Kubernetes，还包括以下几大主题：

- 云原生应用与微服务架构
- 将微服务与Service mesh架构的代码级呈现
- Kubernetes与微服务结合实践

起初写作本书时，安装的所有组件、所用示例和操作等皆基于**Kubernetes1.6+** 版本，同时我们也将密切关注kubernetes的版本更新，随着它的版本更新升级，本书中的kubernetes版本和示例也将随之更新。

GitHub 地址：https://github.com/rootsongjc/kubernetes-handbook

Gitbook 在线浏览：https://jimmysong.io/kubernetes-handbook/

贡献与致谢

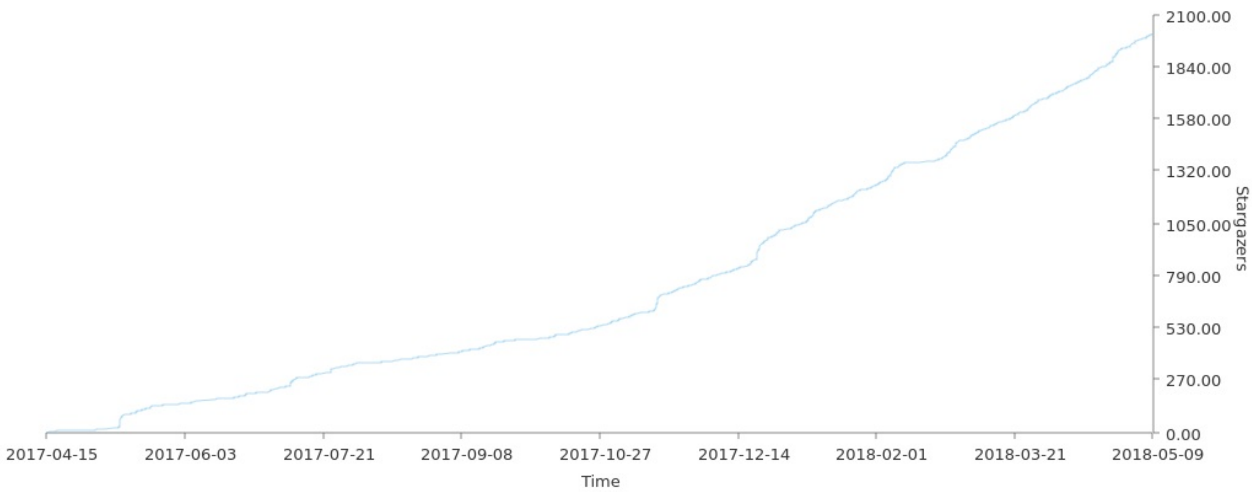
感谢大家对本书做出的贡献！

查看贡献者列表

查看如何贡献

查看文档的组织结构与使用方法

Stargazers over time



图片 - *Stargazers over time*

社区&读者交流

- 微信群：K8S&Cloud Native实战，扫描我的微信二维码，Jimmy Song，或直接搜索微信号*jimmysong*后拉您入群，请增加备注（姓名-公司/学校/博客/社区/研究所/机构等）。
- **Slack**：全球中文用户可以加入Kubernetes官方Slack中文频道**cn-users channel**
- 知乎专栏：云原生应用架构
- 微信公众号：扫描下面的二维码关注微信公众号CloudNativeGo（云原生应用架构）



支持本书

为贡献者加油□！为云原生干杯□！

使用微信扫一扫请贡献者喝一杯

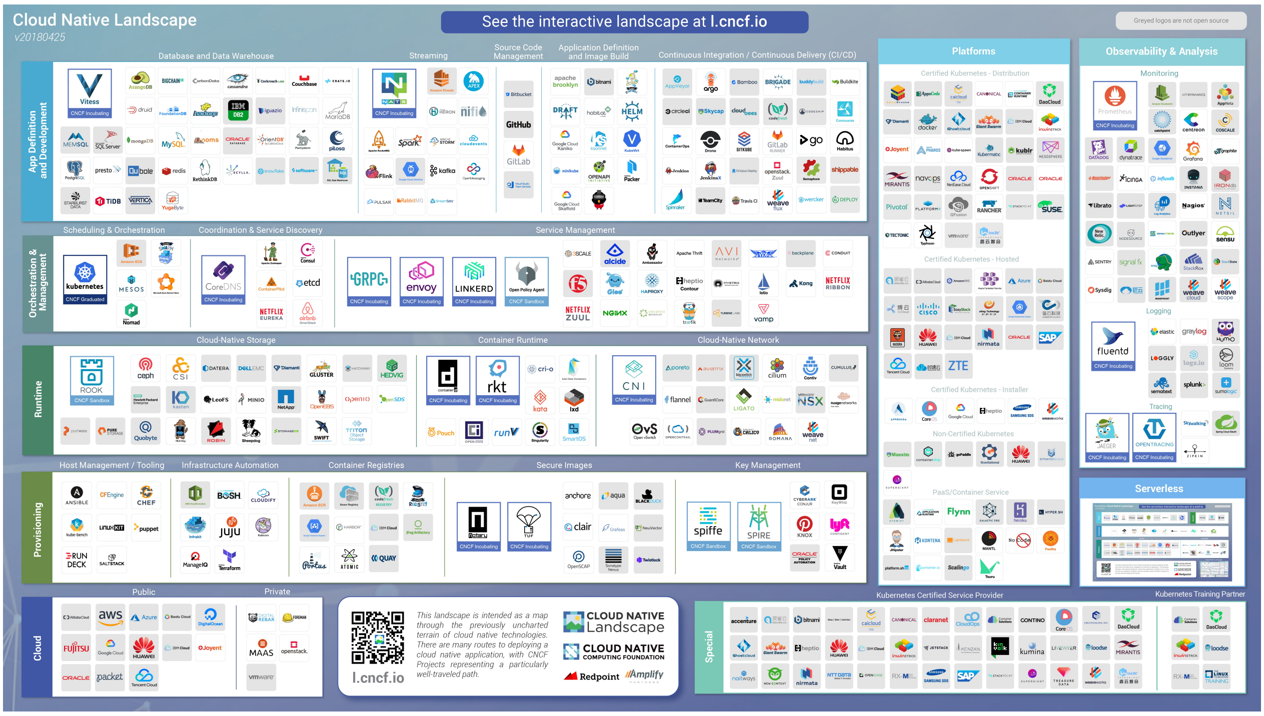


CNCF - 云原生计算基金会简介

CNCF，全称Cloud Native Computing Foundation（云原生计算基金会），口号是坚持和整合开源技术来让编排容器作为微服务架构的一部分，其作为致力于云原生应用推广和普及的一支重要力量，不论您是云原生应用的开发者、管理者还是研究人员都有必要了解。

CNCF作为一个厂商中立的基金会，致力于Github上的快速成长的开源技术的推广，如Kubernetes、Prometheus、Envoy等，帮助开发人员更快更好的构建出色的产品。

下图是CNCF的全景图。



图片 - *CNCF landscape*

其中包含了CNCF中托管的项目，还有很多是非CNCF项目。

关于CNCF的使命与组织方式请参考CNCF宪章，概括的讲CNCF的使命包括以下三点：

- 容器化包装。

- 通过中心编排系统的动态资源管理。
- 面向微服务。

CNCF这个角色的作用是推广技术，形成社区，开源项目管理与推进生态系统健康发展。

另外CNCF组织由以下部分组成：

- 会员：白金、金牌、银牌、最终用户、学术和非赢利成员，不同级别的会员在治理委员会中的投票权不同。
- 理事会：负责事务管理
- **TOC**（技术监督委员会）：技术管理
- 最终用户社区：推动CNCF技术的采纳并选举最终用户技术咨询委员会
- 最终用户技术咨询委员会：为最终用户会议或向理事会提供咨询
- 营销委员会：市场推广

CNCF项目成熟度分级与毕业条件

每个CNCF项目都需要有个成熟度等级，申请成为CNCF项目的时候需要确定项目的成熟度级别。

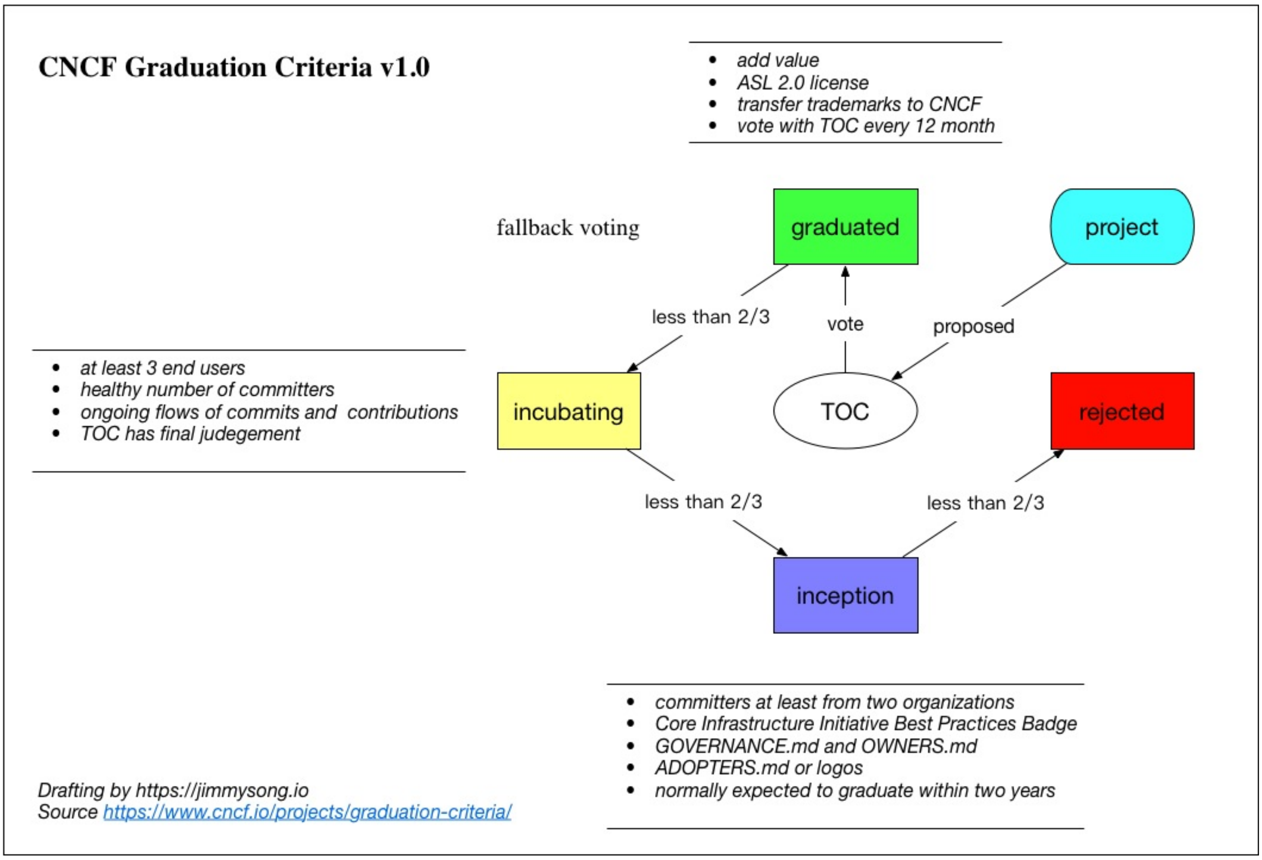
成熟度级别（Maturity Level）包括以下三种：

- inception（初级）
- incubating（孵化中）
- graduated（毕业）

是否可以成为CNCF项目需要通过Technical Oversight Committee (技术监督委员会）简称TOC，投票采取fallback策略，即回退策略，先从最高级别（graduated）开始，如果2/3多数投票通过的话则确认为该级别，如果没通过的话，则进行下一低级别的投票，如果一直到inception级别都没得到2/3多数投票通过的话，则拒绝其进入CNCF项目。

当前所有的CNCF项目可以访问https://www.cncf.io/projects/ 。

项目所达到相应成熟度需要满足的条件和投票机制见下图：



图片 - *CNCF*项目成熟度级别

TOC（技术监督委员会）

TOC（Technical Oversight Committee）作为CNCF中的一个重要组织，它的作用是：

- 定义和维护技术视野
- 审批新项目加入组织，为项目设定概念架构
- 接受最终用户的反馈并映射到项目中
- 调整组件见的访问接口，协调组件之间兼容性

TOC成员通过选举产生，见选举时间表。

参考CNCF TOC：https://github.com/cncf/toc

参考

- https://www.cncf.io

- https://www.cncf.io/projects/graduation-criteria/
- https://www.cncf.io/about/charter/
- https://github.com/cncf/landscape
- https://github.com/cncf/toc

Play with Kubernetes

本书的主角是Kubernetes，在开始后面几章的长篇大论之前让大家可以零基础上手，揭开Kubernetes的神秘面纱。

本文不是讲解Kubernetes的高深原理也不是讲Kuberentes的具体用法，而是通过Play with Kubernetes来带您进入Kubernetes的世界，相当于Kubernetes世界的“Hello World”！而且除了一台可以上网的电脑和浏览器之外不需要再准备任何东西，甚至（至少目前为止）不需要注册账号，上手即玩。

当然免费使用也是有限制的，当前的限制如下：

- 内置kubeadm来创建kubernetes集群，版本为v1.8.4
- 每个实例配置为1 core，4G Memory，最多创建5个实例
- 每个集群的使用时间是4个小时（当然你可以同时启动多个集群，根据浏览器的session来判断集群）
- 在Kubernetes集群中创建的服务无法通过外网访问，只能在Play with Kubernetes的网络内访问

登陆Play with Kubernetes，点击【登陆】-【开始】即可开始你的Kubernetes之旅！

创建Kubernetes集群

启动第一个实例作为Master节点，在web终端上执行：

1. 初始化master节点：

```
kubeadm init --apiserver-advertise-address $(hostname -i)
```

1. 初始化集群网络：

```
kubectl apply -n kube-system -f  "https://cloud.weave.works/k8s/
net?k8s-version=$(kubectl version | base64 | tr -d '\n')"
```

1. 执行下列初始化命令：

```
mkdir -p $HOME/.kube
cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
chown $(id -u):$(id -g) $HOME/.kube/config
```

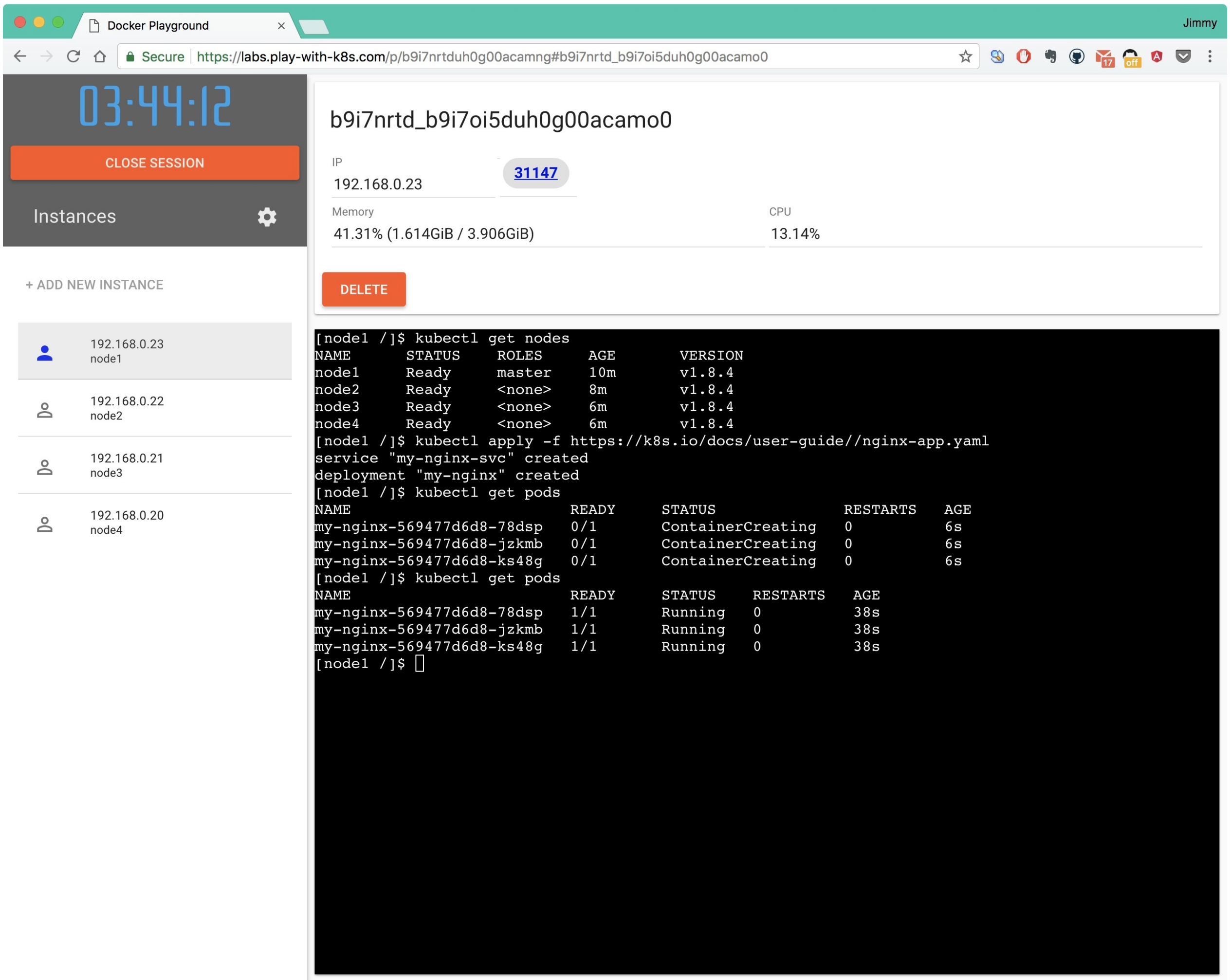
1. 启动新的实例作为node节点，根据master节点上的提示，在新的web终端上执行：

```
kubeadm join --token 513212.cfea0165b8988d18 192.168.0.13:6443 --discovery-token-ca-cert-hash sha256:b7b6dcc98f3ead3f9e363cb3928fbc04774ee0d63e8eb2897ae30e05aebf8070
```

注意：`192.168.0.13` 是master节点的IP，请替换您的master节点的实际IP。

再添加几个实例，重复执行第四步，即可向Kubernetes集群中增加节点。

此时在master节点上执行`kubectl get nodes`查看节点所有节点状态，并创建nginx deployment，如下图所示：



图片 - *Play with Kubernetes*网页截图

Play with Kuberentes (PWK) is a project hacked by Marcos Lilljedahl and Jonathan Leibiusky and sponsored by Docker Inc.

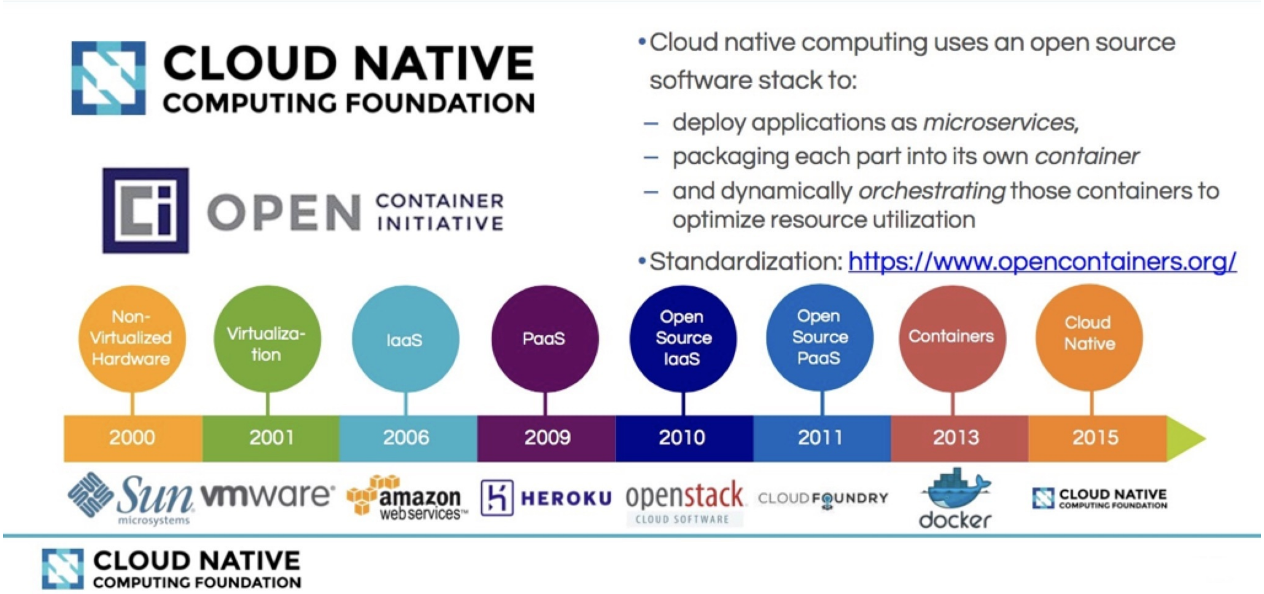
Kubernetes与云原生应用概览

几个月前Mesos已经宣布支持kubernetes，而在2017年10月份的DockerCon EU上Docker公司宣布同时官方支持Swarm和Kubernetes容器编排，kubernetes已然成为容器编排调度的标准。

作为全书的开头，首先从历史、生态和应用角度介绍一下kubernetes与云原生应用，深入浅出，高屋建瓴，没有深入到具体细节，主要是为了给初次接触kubernetes的小白扫盲，具体细节请参考链接。

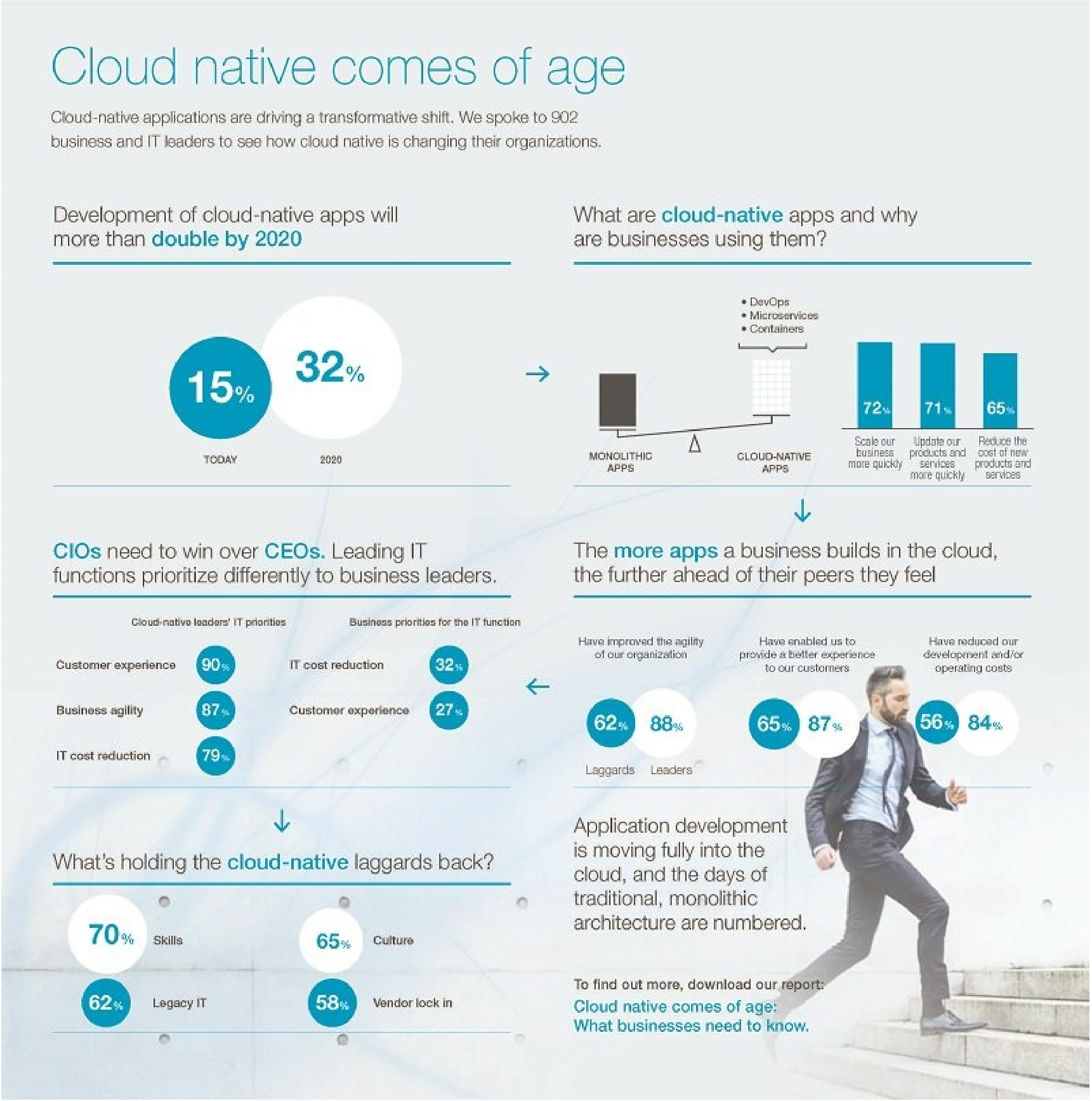
从云计算到微服务再到云原生计算

下面将从云计算的发展历程引入云原生计算，请先看下图：



图片 - 云计算演进历程

云原生应用到2020年将比目前至少翻一番，下图是Marc Wilczek的调查报告。



图片 - 来自 *Twitter @MarcWilczek*

云计算介绍

云计算包含的内容十分繁杂，也有很多技术和公司牵强附会说自己是云计算公司，说自己是做云的，实际上可能风马牛不相及。说白了，云计算就是一种配置资源的方式，根据资源配置方式的不同我们可以把云计算从宏观上分为以下三种类型：

- IaaS：这是为了想要建立自己的商业模式并进行自定义的客户，例如亚马逊的EC2、S3存储、Rackspace虚拟机等都是IaaS。
- PaaS：工具和服务的集合，对于想用它来构建自己的应用程序或者想快速得

将应用程序部署到生产环境而不必关心底层硬件的用户和开发者来说是特别有用的，比如Cloud Foundry、Google App Engine、Heroku等。
- SaaS：终端用户可以直接使用的应用程序。这个就太多，我们生活中用到的很多软件都是SaaS服务，只要基于互联网来提供的服务基本都是SaaS服务，有的服务是免费的，比如Google Docs，还有更多的是根据我们购买的Plan和使用量付费，比如GitHub、各种云存储。

微服务介绍

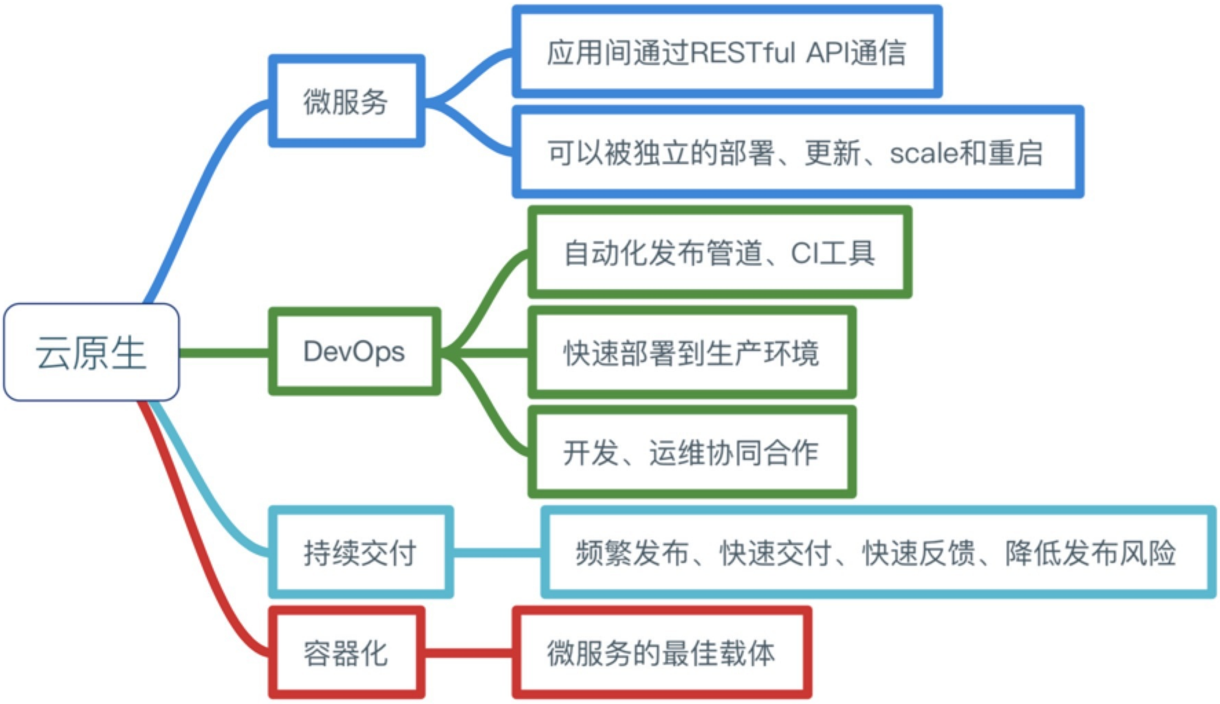
微服务（Microservices）这个词比较新颖，但是其实这种架构设计理念早就有了。微服务是一种分布式架构设计理念，为了推动细粒度服务的使用，这些服务要能协同工作，每个服务都有自己的生命周期。一个微服务就是一个独立的实体，可以独立的部署在PAAS平台上，也可以作为一个独立的进程在主机中运行。服务之间通过API访问，修改一个服务不会影响其它服务。

要想了解微服务的详细内容推荐阅读《微服务设计》（Sam Newman著），我写过这本书的读书笔记 - 微服务设计读书笔记。

下文中会谈到kubernetes与微服务的关系，其中kubernetes的service天生就适合于微服务。

云原生概念介绍

下面是Cloud Native概念思维导图



图片 - *Cloud native*思维导图

云原生准确来说是一种文化，更是一种潮流，它是云计算的一个必然导向。它的意义在于让云成为云化战略成功的基石，而不是阻碍，如果业务应用上云之后开发和运维人员比原先还痛苦，成本还高的话，这样的云我们宁愿不上。

自从云的概念开始普及，许多公司都部署了实施云化的策略，纷纷搭建起云平台，希望完成传统应用到云端的迁移。但是这个过程中会遇到一些技术难题，上云以后，效率并没有变得奇高，故障也没有迅速定位。

为了解决传统应用升级缓慢、架构臃肿、不能快速迭代、故障不能快速定位、问题无法快速解决等问题，云原生这一概念横空出世。云原生可以改进应用开发的效率，改变企业的组织结构，甚至会在文化层面上直接影响一个公司的决策。

另外，云原生也很好地解释了云上运行的应用应该具备什么样的架构特性——敏捷性、可扩展性、故障可恢复性。

综上所述，云原生应用应该具备以下几个关键词：

- 敏捷
- 可靠

- 高弹性
- 易扩展
- 故障隔离保护
- 不中断业务持续更新

以上特性也是云原生区别于传统云应用的优势特点。

从宏观概念上讲，云原生是不同思想的集合，集目前各种热门技术之大成，具体包括如下图所示的几个部分。

Kubernetes与云原生的关系

Kuberentes可以说是乘着docker和微服务的东风，一经推出便迅速蹿红，它的很多设计思想都契合了微服务和云原生应用的设计法则，这其中最著名的就是开发了Heroku PaaS平台的工程师们总结的 Twelve-factor App了。

下面我将讲解Kubernetes设计时是如何按照了十二因素应用法则，并给出kubernetes中的应用示例，并附上一句话简短的介绍。

Kubernetes介绍

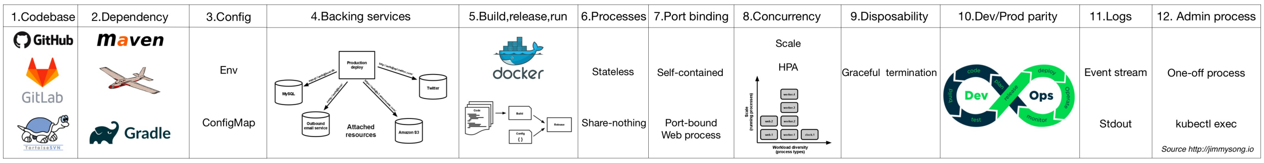
Kubernetes是Google基于Borg开源的容器编排调度引擎，作为CNCF（Cloud Native Computing Foundation）最重要的组件之一，它的目标不仅仅是一个编排系统，而是提供一个规范，可以让你来描述集群的架构，定义服务的最终状态，kubernetes可以帮你将系统自动得达到和维持在这个状态。

更直白的说，Kubernetes用户可以通过编写一个yaml或者json格式的配置文件，也可以通过工具/代码生成或直接请求kubernetes API创建应用，该配置文件中包含了用户想要应用程序保持的状态，不论整个kubernetes集群中的个别主机发生什么问题，都不会影响应用程序的状态，你还可以通过改变该配置文件或请求kubernetes API来改变应用程序的状态。

12因素应用

12因素应用提出已经有几年的时间了，每个人对其可能都有自己的理解，切不可生搬硬套，也不一定所有云原生应用都必须符合这12条法则，其中有几条法则可能还有点争议，有人对其的解释和看法不同。

大家不要孤立的来看这每一个因素，将其与自己软件开发流程联系起来，这12个因素大致就是按照软件从开发到交付的流程顺序来写的。



图片 - 十二因素应用

1.基准代码

每个代码仓库（repo）都生成docker image保存到镜像仓库中，并使用唯一的ID管理，在Jenkins中使用编译时的ID。

2.依赖

显式得声明代码中的依赖，使用软件包管理工具声明，比如Go中的Glide。

3.配置

将配置与代码分离，应用部署到kubernetes中可以使用容器的环境变量或ConfigMap挂载到容器中。

4.后端服务

把后端服务当作附加资源，实质上是计算存储分离和降低服务耦合，分解单体应用。

5.构建、发布、运行

严格分离构建和运行，每次修改代码生成新的镜像，重新发布，不能直接修改运行时的代码和配置。

6.进程

应用程序进程应该是无状态的，这意味着再次重启后还可以计算出原先的状态。

7.端口绑定

在kubernetes中每个Pod都有独立的IP，每个运行在Pod中的应用不必关心端口是否重复，只需在service中指定端口，集群内的service通过配置互相发现。

8.并发

每个容器都是一个进程，通过增加容器的副本数实现并发。

9.易处理

快速启动和优雅终止可最大化健壮性，kuberentes优秀的Pod生存周期控制。

10.开发环境与线上环境等价

在kubernetes中可以创建多个namespace，使用相同的镜像可以很方便的复制一套环境出来，镜像的使用可以很方便的部署一个后端服务。

11.日志

把日志当作事件流，使用stdout输出并收集汇聚起来，例如到ES中统一查看。

12.管理进程

后台管理任务当作一次性进程运行，`kubectl exec` 进入容器内部操作。

另外，Cloud Native Go 这本书的作者，CapitalOne公司的Kevin Hoffman在TalkingData T11峰会上的High Level Cloud Native的演讲中讲述了云原生应用的15个因素，在原先的12因素应用的基础上又增加了如下三个因素：

API优先

- 服务间的合约
- 团队协作的规约
- 文档化、规范化
- RESTful或RPC

监控

- 实时监控远程应用
- 应用性能监控（APM）
- 应用健康监控
- 系统日志
- 不建议在线Debug

认证授权

- 不要等最后才去考虑应用的安全性
- 详细设计、明确声明、文档化

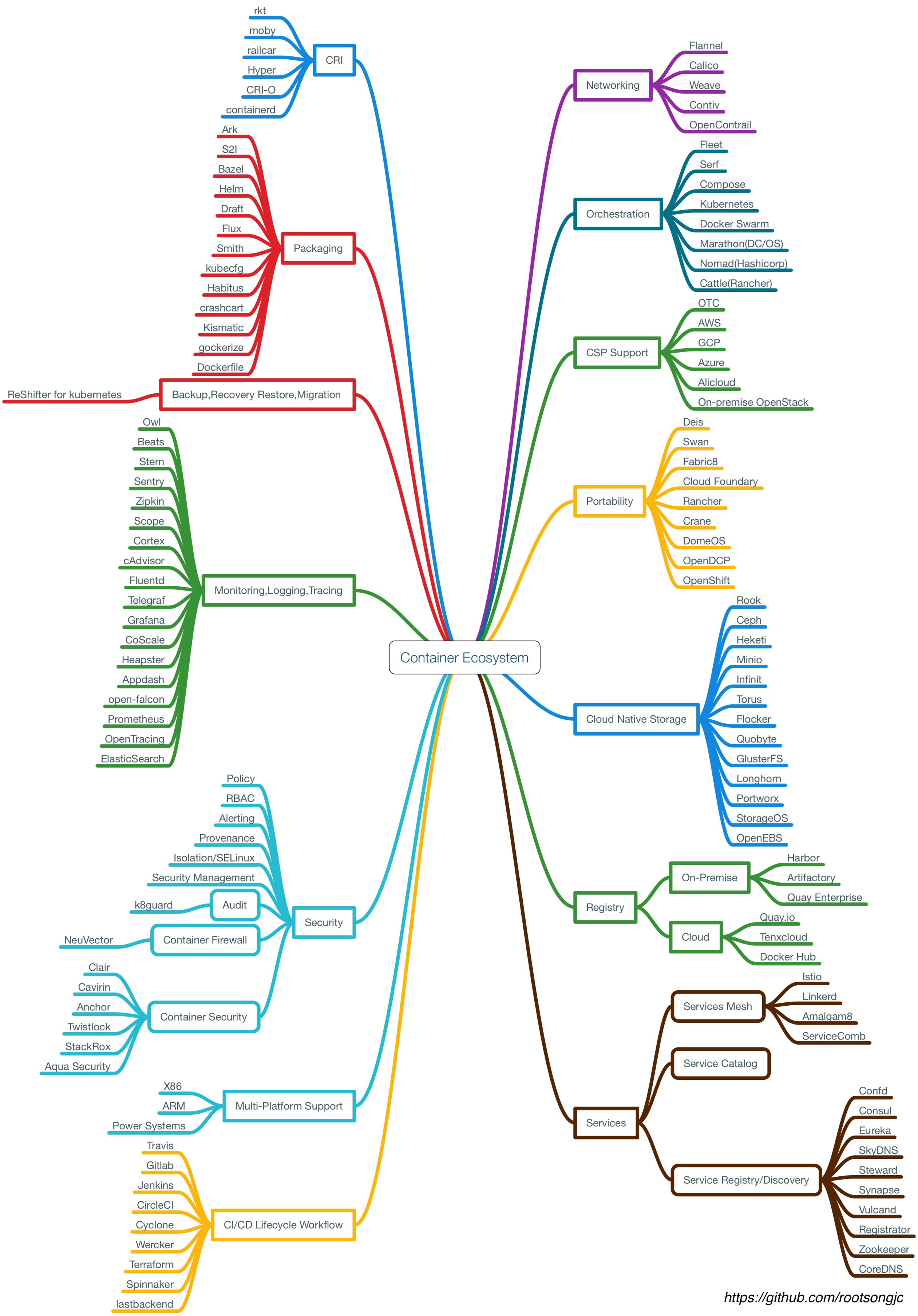
- Bearer token、OAuth、OIDC认证
- 操作审计

详见High Level Cloud Native From Kevin Hoffman。

Kubernetes中的资源管理与容器设计模式

Kubernetes通过声明式配置，真正让开发人员能够理解应用的状态，并通过同一份配置可以立马启动一个一模一样的环境，大大提高了应用开发和部署的效率，其中kubernetes设计的多种资源类型可以帮助我们定义应用的运行状态，并使用资源配置来细粒度得明确限制应用的资源使用。

而容器生态的成熟是 Kubernetes 诞生的前提，在谈到容器的设计模式之前我们先来了解下容器生态，请看下图：


Container Ecosystem
CRI
rkt
moby
railcar
Hyper
CRI-O
containerd
Packaging
Ark
S2I
Bazel
Helm
Draft
Flux
Smith
kubecfg
Habitus
crashcart
Kismatic
gockerize
Dockerfile
Backup,Recovery Restore,Migration
ReShifter for kubernetes
Monitoring,Logging,Tracing
Owl
Beats
Stern
Sentry
Zipkin
Scope
Cortex
cAdvisor
Fluentd
Telegraf
Grafana
CoScale
Heapster
Appdash
open-falcon
Prometheus
OpenTracing
ElasticSearch
Security
Policy
RBAC
Alerting
Provenance
Isolation/SELinux
Security Management
Audit
k8guard
Container Firewall
NeuVector
Container Security
Clair
Cavirin
Anchor
Twistlock
StackRox
Aqua Security
Multi-Platform Support
X86
ARM
Power Systems
CI/CD Lifecycle Workflow
Travis
Gitlab
Jenkins
CircleCI
Cyclone
Wercker
Terraform
Spinnaker
lastbackend
Networking
Flannel
Calico
Weave
Contiv
OpenContrail
Orchestration
Fleet
Serf
Compose
Kubernetes
Docker Swarm
Marathon(DC/OS)
Nomad(Hashicorp)
Cattle(Rancher)
CSP Support
OTC
AWS
GCP
Azure
Alicloud
On-premise OpenStack
Portability
Deis
Swan
Fabric8
Cloud Foundary
Rancher
Crane
DomeOS
OpenDCP
OpenShift
Cloud Native Storage
Rook
Ceph
Heketi
Minio
Infinit
Torus
Flocker
Quobyte
GlusterFS
Longhorn
Portworx
StorageOS
OpenEBS
Registry
On-Premise
Harbor
Artifactory
Quay Enterprise
Cloud
Quay.io
Tenxcloud
Docker Hub
Services
Services Mesh
Istio
Linkerd
Amalgam8
ServiceComb
Service Catalog
Service Registry/Discovery
Confd
Consul
Eureka
SkyDNS
Steward
Synapse
Vulcand
Registrator
Zookeeper
CoreDNS
https://github.com/rootsongjc

图片 - 容器生态

关于 Docker 容器的更多内容请参考 Docker最佳实践。

容器的设计模式

Kubernetes提供了多种资源对象，用户可以根据自己应用的特性加以选择。这些对象有：

类别	名称
资源对象	Pod、ReplicaSet、ReplicationController、Deployment、StatefulSet、DaemonSet、Job、CronJob、HorizontalPodAutoscaling
配置对象	Node、Namespace、Service、Secret、ConfigMap、Ingress、Label、ThirdPartyResource、 ServiceAccount
存储对象	Volume、Persistent Volume
策略对象	SecurityContext、ResourceQuota、LimitRange

在 Kubernetes 系统中，*Kubernetes* 对象 是持久化的条目。Kubernetes 使用这些条目去表示整个集群的状态。特别地，它们描述了如下信息：

- 什么容器化应用在运行（以及在哪个 Node 上）
- 可以被应用使用的资源
- 关于应用如何表现的策略，比如重启策略、升级策略，以及容错策略

Kubernetes 对象是“目标性记录”—— 一旦创建对象，Kubernetes 系统将持续工作以确保对象存在。通过创建对象，可以有效地告知 Kubernetes 系统，所需要的集群工作负载看起来是什么样子的，这就是 Kubernetes 集群的 期望状态。

详见Kubernetes Handbook - Objects。

资源限制与配额

两层的资源限制与配置

- Pod级别，最小的资源调度单位
- Namespace级别，限制资源配额和每个Pod的资源使用区间

请参考Kubernetes中的ResourceQuota和LimitRange配置资源限额

管理Kubernetes集群

手工部署Kubernetes是一个很艰巨的活，你需要了解网络配置、docker的安装与使用、镜像仓库的构建、角色证书的创建、kubernetes的基本原理和构成、kubernetes应用程序的yaml文件编写等。

我编写了一本kubernetes-handbook可供大家免费阅读，该书记录了本人从零开始学习和使用Kubernetes的心路历程，着重于经验分享和总结，同时也会有相关的概念解析，希望能够帮助大家少踩坑，少走弯路。

部署Kubernetes集群

使用二进制部署 `kubernetes` 集群的所有组件和插件，而不是使用 `kubeadm` 等自动化方式来部署集群，同时开启了集群的TLS安全认证，这样可以帮助我们解系统各组件的交互原理，进而能快速解决实际问题。详见在CentOS上部署Kubernetes集群。

集群详情

- Kubernetes 1.6.0
- Docker 1.12.5（使用yum安装）
- Etcd 3.1.5
- Flanneld 0.7 vxlan 网络
- TLS 认证通信 (所有组件，如 etcd、kubernetes master 和 node)
- RBAC 授权
- kublet TLS BootStrapping
- kubedns、dashboard、heapster(influxdb、grafana)、EFK(elasticsearch、fluentd、kibana) 集群插件
- 私有docker镜像仓库harbor（请自行部署，harbor提供离线安装包，直接使用docker-compose启动即可）

步骤介绍

1. 创建 TLS 证书和秘钥
2. 创建kubeconfig文件
3. 创建高可用etcd集群
4. 安装kubectl命令行工具
5. 部署master节点
6. 安装flannel网络插件
7. 部署node节点
8. 安装kubedns插件
9. 安装dashboard插件
10. 安装heapster插件
11. 安装EFK插件

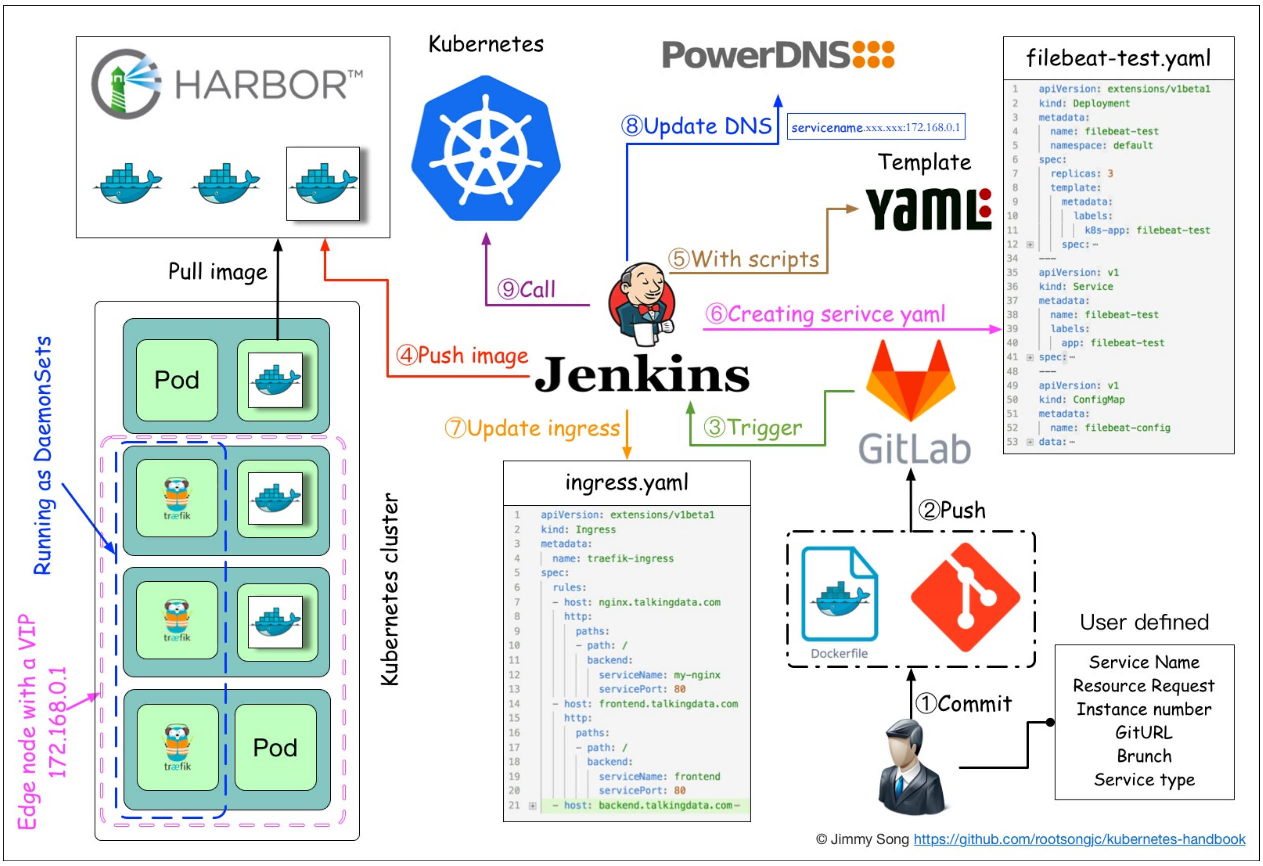
服务发现与负载均衡

Kubernetes在设计之初就充分考虑了针对容器的服务发现与负载均衡机制，提供了Service资源，并通过kube-proxy配合cloud provider来适应不同的应用场景。随着kubernetes用户的激增，用户场景的不断丰富，又产生了一些新的负载均衡机制。目前，kubernetes中的负载均衡大致可以分为以下几种机制，每种机制都有其特定的应用场景：

- **Service**：直接用Service提供cluster内部的负载均衡，并借助cloud provider提供的LB提供外部访问
- **Ingress**：还是用Service提供cluster内部的负载均衡，但是通过自定义LB提供外部访问
- **Service Load Balancer**：把load balancer直接跑在容器中，实现Bare Metal的Service Load Balancer
- **Custom Load Balancer**：自定义负载均衡，并替代kube-proxy，一般在物理部署Kubernetes时使用，方便接入公司已有的外部服务

详见Kubernetes Handbook - 服务发现与负载均衡。

持续集成与发布



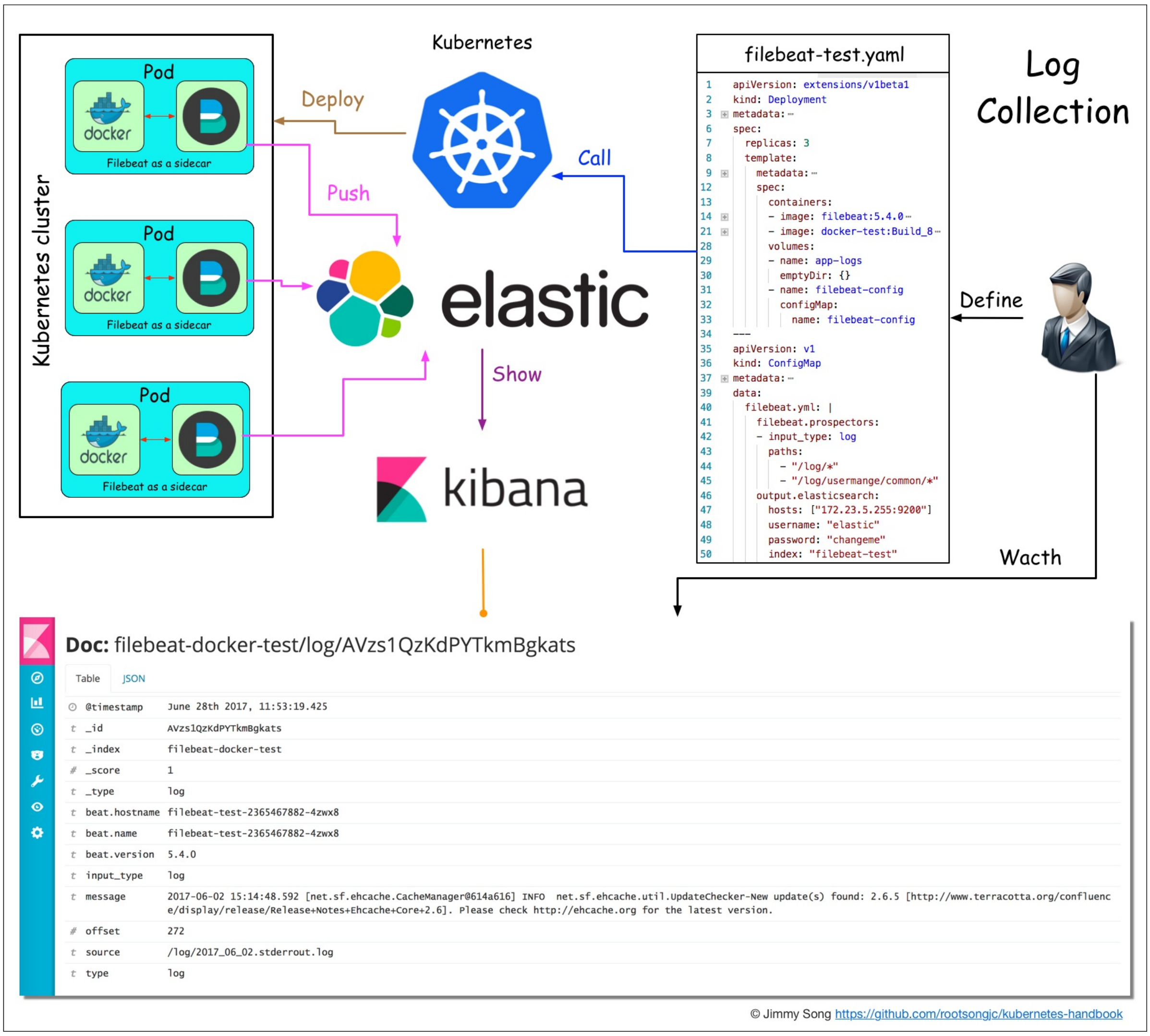
图片 - 使用*Jenkins*进行持续集成与发布流程图

应用构建和发布流程说明：

1. 用户向Gitlab提交代码，代码中必须包含 `Dockerfile`
2. 将代码提交到远程仓库
3. 用户在发布应用时需要填写git仓库地址和分支、服务类型、服务名称、资源数量、实例个数，确定后触发Jenkins自动构建
4. Jenkins的CI流水线自动编译代码并打包成docker镜像推送到Harbor镜像仓库
5. Jenkins的CI流水线中包括了自定义脚本，根据我们已准备好的kubernetes的YAML模板，将其中的变量替换成用户输入的选项
6. 生成应用的kubernetes YAML配置文件
7. 更新Ingress的配置，根据新部署的应用的名称，在ingress的配置文件中增加一条路由信息
8. 更新PowerDNS，向其中插入一条DNS记录，IP地址是边缘节点的IP地址。关于边缘节点，请查看边缘节点配置
9. Jenkins调用kubernetes的API，部署应用

日志收集与监控

基于现有的ELK日志收集方案，稍作改造，选用filebeat来收集日志，可以作为sidecar的形式跟应用运行在同一个Pod中，比较轻量级消耗资源比较少。



图片 - *filebeat*日志收集架构图

详见Kubernetes Handbook - 应用日志收集。

安全性与权限管理

Kubernetes是一个多租户的云平台，因此必须对用户的权限加以限制，对用户空间进行隔离。Kubernetes中的隔离主要包括这几种：

- 网络隔离：需要使用网络插件，比如calico。

- 资源隔离：kubernetes原生支持资源隔离，pod就是资源就是隔离和调度的最小单位，同时使用namespace限制用户空间和资源限额。
- 身份隔离：使用RBAC-基于角色的访问控制，多租户的身份认证和权限控制。

如何开发Kubernetes原生应用步骤介绍

当我们有了一个kubernetes集群后，如何在上面开发和部署应用，应该遵循怎样的流程？下面我将展示如何使用go语言开发和部署一个kubernetes native应用，使用wercker进行持续集成与持续发布，我将以一个很简单的前后端访问，获取伪造数据并展示的例子来说明。

云原生应用开发示例

我们将按照如下步骤来开发部署一个kubernetes原生应用并将它部署到kubernetes集群上开放给集群外访问：

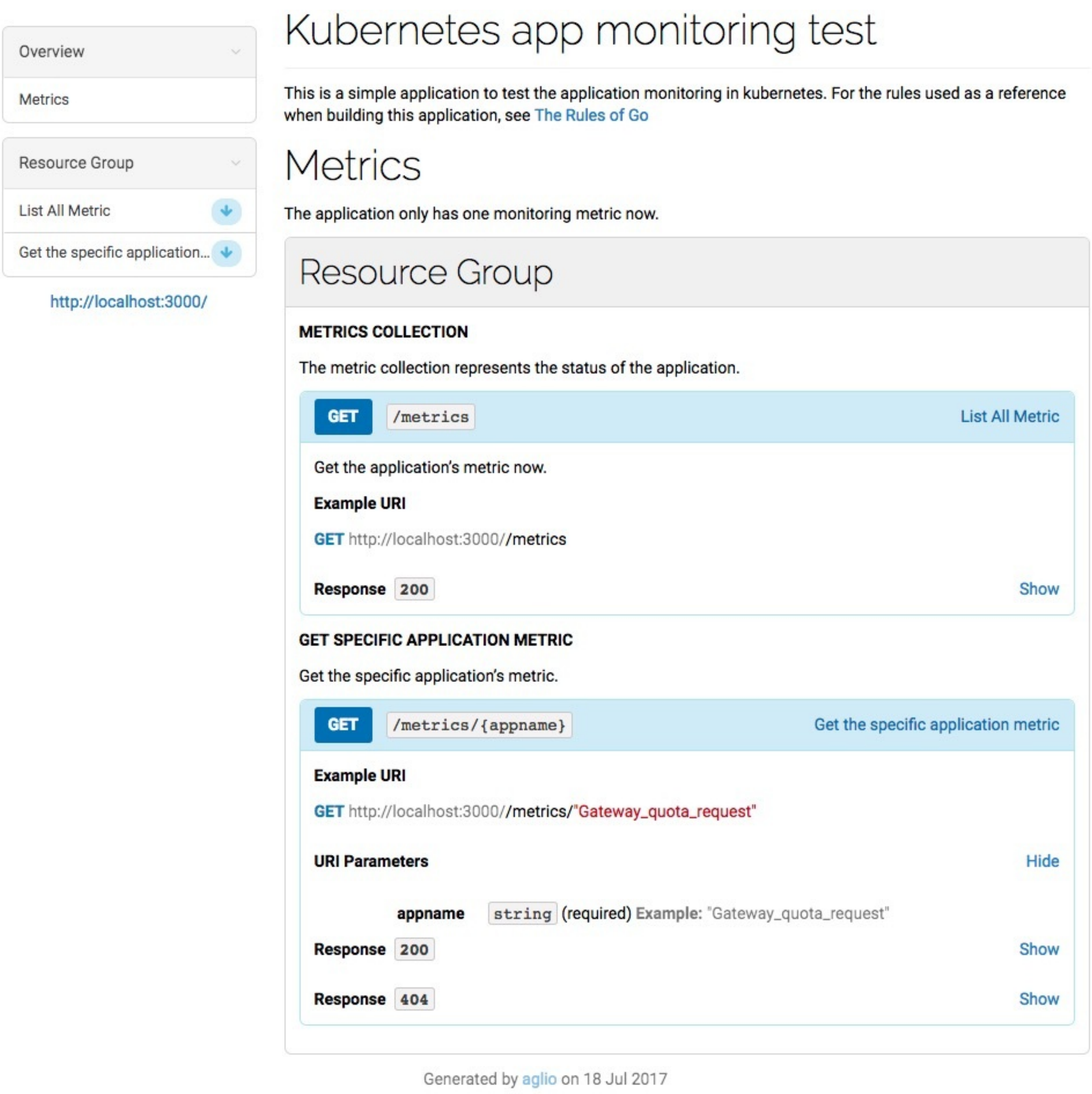
1. 服务API的定义
2. 使用Go语言开发kubernetes原生应用
3. 一个持续构建与发布工具与环境
4. 使用traefik和VIP做边缘节点提供外部访问路由

我写了两个示例用于演示，开发部署一个伪造的 metric 并显示在 web 页面上，包括两个service：

- k8s-app-monitor-test：生成模拟的监控数据，发送http请求，获取json返回值
- K8s-app-monitor-agent：获取监控数据并绘图，访问浏览器获取图表

定义**API**生成**API**文档

使用 `API blueprint` 格式，定义API文档，格式类似于markdown，再使用aglio生成HTML文档。



图片 - *API*文档

详见：如何开发部署kubernetes native应用。

如何迁移到云原生应用架构

Pivotal 是云原生应用的提出者，并推出了 Pivotal Cloud Foundry 云原生应用平台和 Spring 开源 Java 开发框架，成为云原生应用架构中先驱者和探路者。

原书作于2015年，其中的示例主要针对 Java 应用，实际上也适用于任何应用类型，云原生应用架构适用于异构语言的程序开发，不仅仅是针对 Java 语言的程序开发。截止到本人翻译本书时，云原生应用生态系统已经初具规模，CNCF 成员不断发展壮大，基于 Cloud Native 的创业公司不断涌现，kubernetes 引领容器编排潮流，和 Service Mesh 技术（如 Linkerd 和 Istio） 的出现，Go 语言的兴起（参考另一本书 Cloud Native Go）等为我们将应用迁移到云原生架构的提供了更多的方案选择。

迁移到云原生应用架构指南

指出了迁移到云原生应用架构需要做出的企业文化、组织架构和技术变革，并给出了迁移指南。

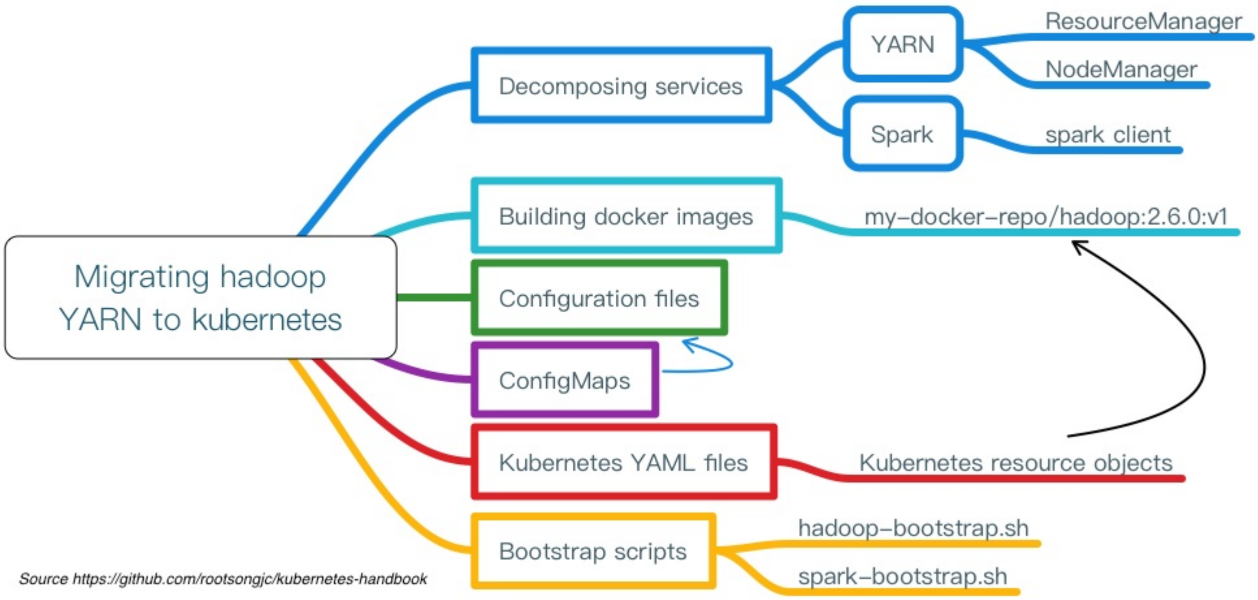
主要讨论的应用程序架构包括：

- 十二因素应用程序：云原生应用程序架构模式的集合
- 微服务：独立部署的服务，只做一件事情
- 自助服务的敏捷基础设施：快速，可重复和一致地提供应用环境和后台服务的平台
- 基于API的协作：发布和版本化的API，允许在云原生应用程序架构中的服务之间进行交互
- 抗压性：根据压力变强的系统

详见：迁移到云原生应用架构

迁移案例解析

迁移步骤示意图如下：



图片 - 迁移步骤示意图

步骤说明：

1. 将原有应用拆解为服务
2. 容器化、制作镜像
3. 准备应用配置文件
4. 准备kubernetes YAML文件
5. 编写bootstarp脚本
6. 创建ConfigMaps

详见：迁移传统应用到Kubernetes步骤详解——以Hadoop YARN为例。

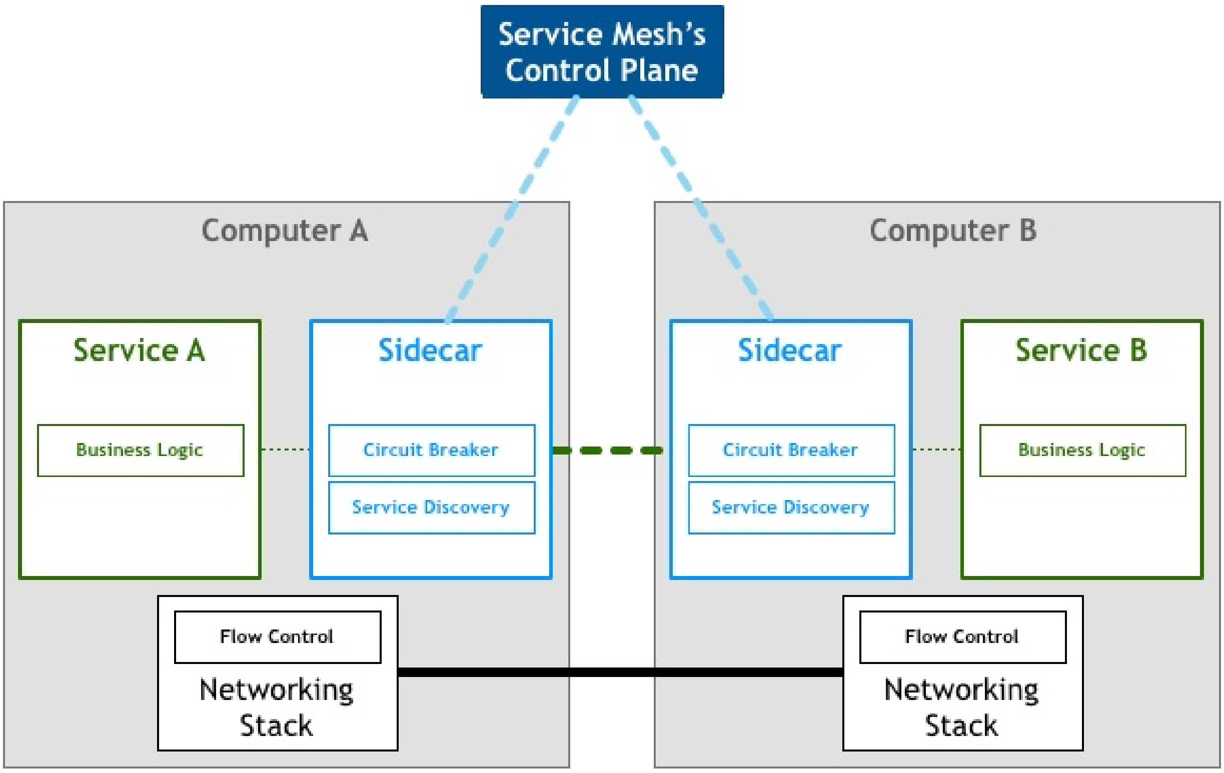
Service mesh基本原理和示例介绍

Service mesh现在一般被翻译作服务网格，目前主流的Service mesh有如下两款：

- Istio：IBM、Google、Lyft共同开源，详细文档见Istio官方文档中文版
- Linkerd：原Twitter工程师开发，现为CNCF中的项目之一

什么是Service mesh

如果用一句话来解释什么是 Service Mesh，可以将它比作是应用程序或者说微服务间的 TCP/IP，负责服务之间的网络调用、限流、熔断和监控。对于编写应用程序来说一般无须关心 TCP/IP 这一层（比如通过 HTTP 协议的 RESTful 应用），同样使用 Service Mesh 也就无须关系服务之间的那些原来是通过应用程序或者其他框架实现的事情，比如 Spring Cloud、OSS，现在只要交给 Service Mesh 就可以了。



图片 - *service mesh*架构图

详见什么是 service mesh - jimmysong.io。

Service mesh使用指南

两款Service mesh各有千秋，我分别写了他们的使用案例指南：

- 微服务管理框架service mesh——Linkerd安装试用笔记
- 微服务管理框架service mesh——Istio安装试用笔记

更多关于 Service Mesh 的内容请访问 Service Mesh 中文网。

使用案例

Kubernetes作为云原生计算的基本组件之一，开源2年时间以来热度与日俱增，它可以跟我们的生产结合，擦出很多火花，比如FaaS和Serverless类应用，都很适合运行在kubernetes上。

关于Cloud Native开源软件生态请参考 Awesome Cloud Native - jimmysong.io。

DevOps

下面是社区中kubernetes开源爱好者的分享内容，我觉得是对kubernetes在DevOps中应用的很好的形式值得大家借鉴。

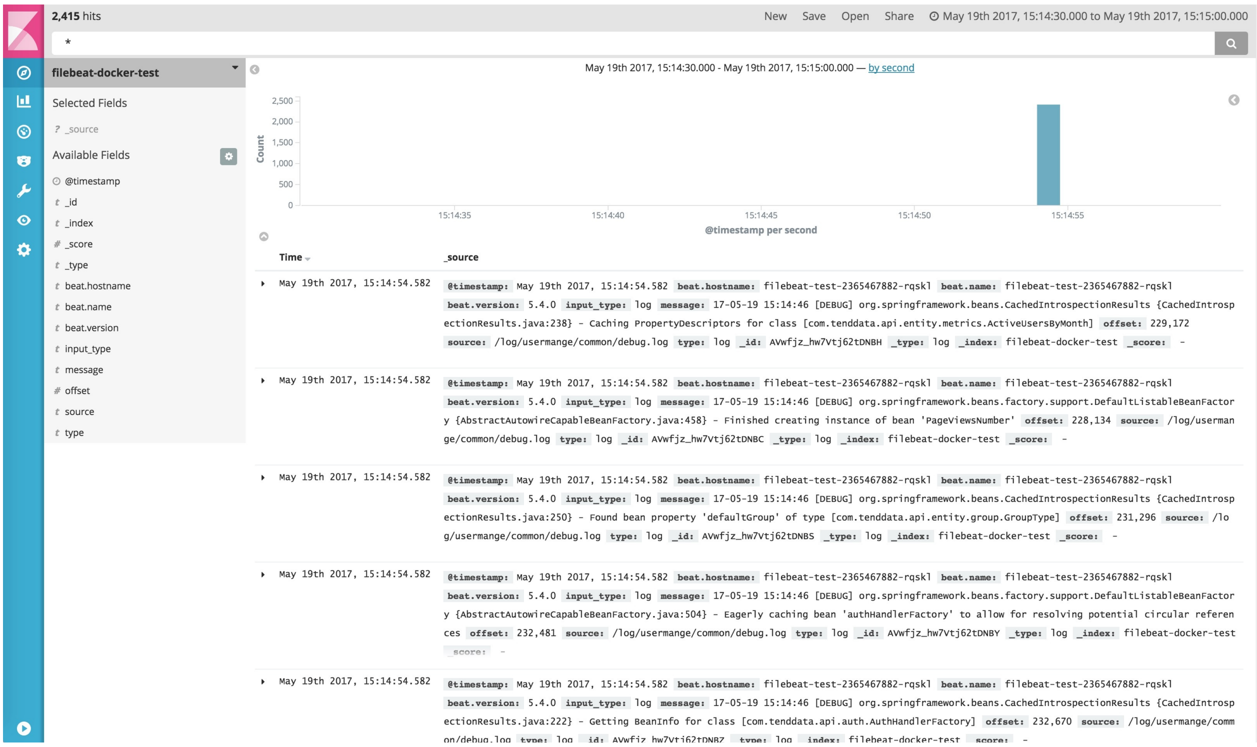
真正践行DevOps，让开发人员在掌握自己的开发和测试环境，让环境一致，让开发效率提升，让运维没有堆积如山的tickets，让监控更加精准，从kubernetes平台开始。

行动指南

1. 根据环境（比如开发、测试、生产）划分 `namespace` ，也可以根据项目来划分
2. 再为每个用户划分一个 `namespace` 、创建一个 `serviceaccount` 和 `kubeconfig` 文件，不同 `namespace` 间的资源隔离，目前不隔离网络，不同 `namespace` 间的服务可以互相访问
3. 创建yaml模板，降低编写kubernetes yaml文件编写难度
4. 在 `kubectl` 命令上再封装一层，增加用户身份设置和环境初始化操作，简化 `kubectl` 命令和常用功能
5. 管理员通过dashboard查看不同 `namespace` 的状态，也可以使用它来使操作更便捷
6. 所有应用的日志统一收集到ElasticSearch中，统一日志访问入口
7. 可以通过Grafana查看所有namespace中的应用的状态和kubernetes集群本身的状态
8. 需要持久化的数据保存在分布式存储中，例如GlusterFS或Ceph中

使用**Kibana**查看日志

日志字段中包括了应用的标签、容器名称、主机名称、宿主机名称、IP地址、时间、

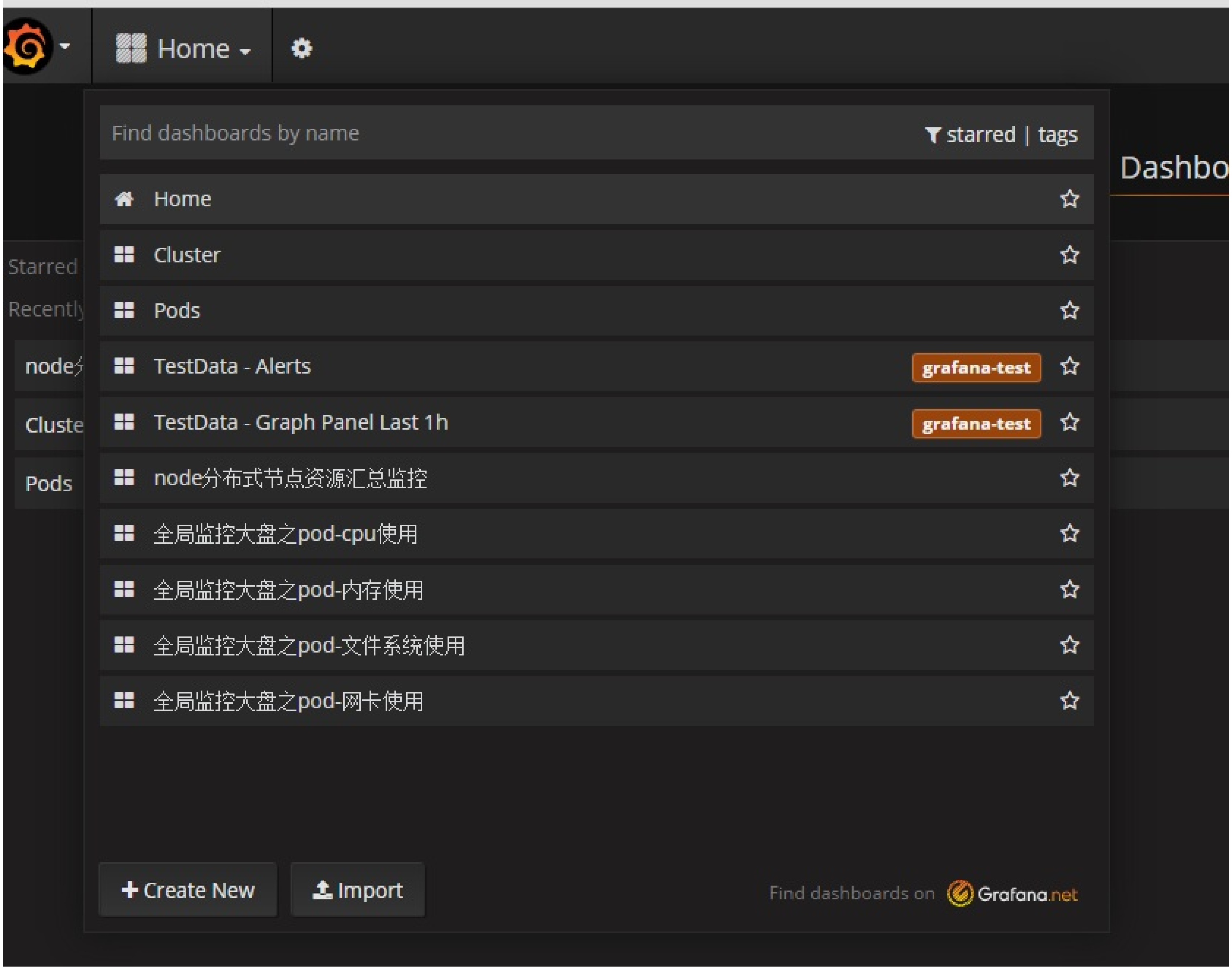


图片 - *kibana*界面

使用**Grafana**查看应用状态

注：感谢【K8SCloud Native 实战群】尊贵的黄金会员小刚同学提供下面的Grafana监控图

监控分类示意图：



图片 - *Grafana*界面示意图*1*

Kubernetes集群全局监控图1

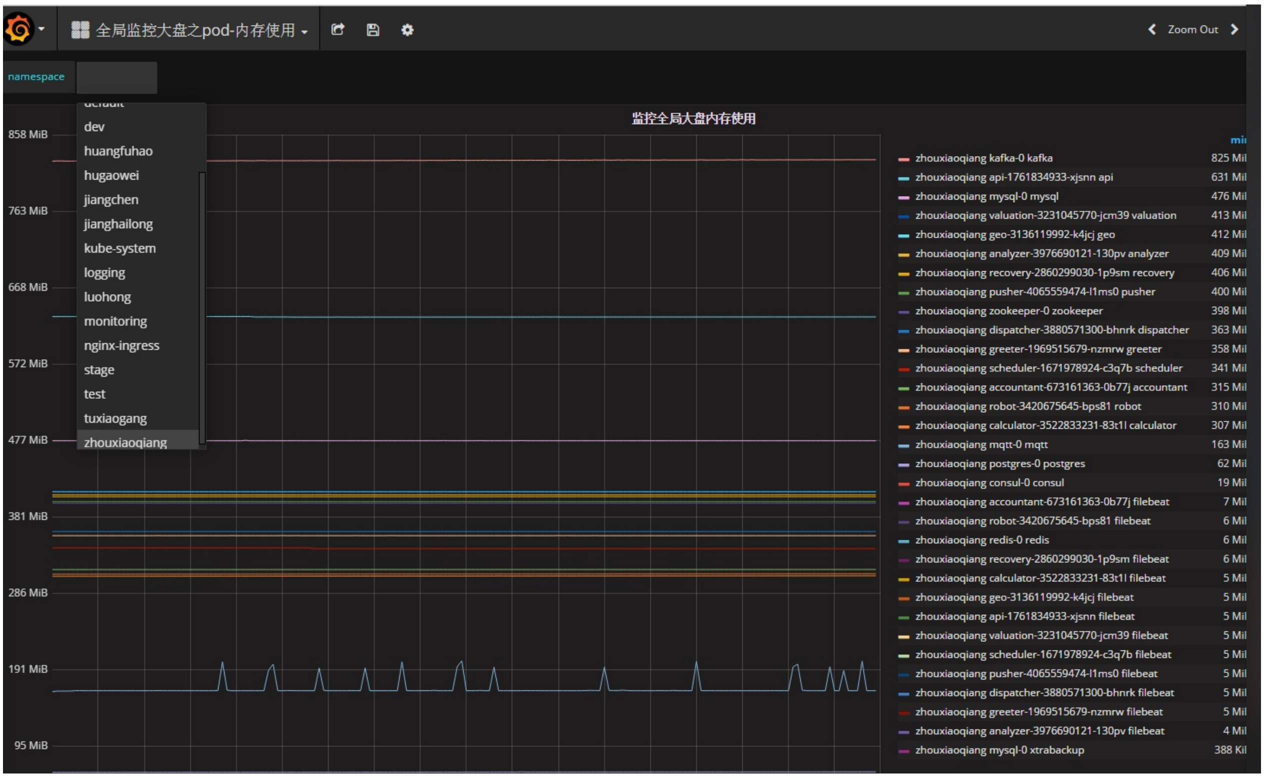
该监控图可以看到集群硬件使用情况。



图片 - *Grafana*界面示意图*2*

Kubernetes全局监控图2

该监控可以看到单个用户的namespace下的所有资源的使用情况。



图片 - *Grafana*界面示意图*3*

Spark on Kubernetes

TL;DR https://jimmysong.io/spark-on-k8s

Spark原生支持standalone、mesos和YARN资源调度，现已支持Kubernetes原生调度，详见运行支持kubernetes原生调度的spark程序-Spark on Kubernetes。

为何要使用**spark on kubernetes**

使用kubernetes原生调度的spark on kubernetes是对原先的spark on yarn和yarn on docker的改变是革命性的，主要表现在以下几点：

1. **Kubernetes**原生调度：不再需要二层调度，直接使用kubernetes的资源调度功能，跟其他应用共用整个kubernetes管理的资源池；
2. 资源隔离，粒度更细：原先yarn中的queue在spark on kubernetes中已不存在，取而代之的是kubernetes中原生的namespace，可以为每个用户分别指定一个namespace，限制用户的资源quota；
3. 细粒度的资源分配：可以给每个spark任务指定资源限制，实际指定多少资源就使用多少资源，因为没有了像yarn那样的二层调度（圈地式的），所以可以更高效和细粒度的使用资源；

4. 监控的变革：因为做到了细粒度的资源分配，所以可以对用户提交的每一个任务做到资源使用的监控，从而判断用户的资源使用情况，所有的metric都记录在数据库中，甚至可以为每个用户的每次任务提交计量；
5. 日志的变革：用户不再通过yarn的web页面来查看任务状态，而是通过pod的log来查看，可将所有的kuberentes中的应用的日志等同看待收集起来，然后可以根据标签查看对应应用的日志；

如何提交任务

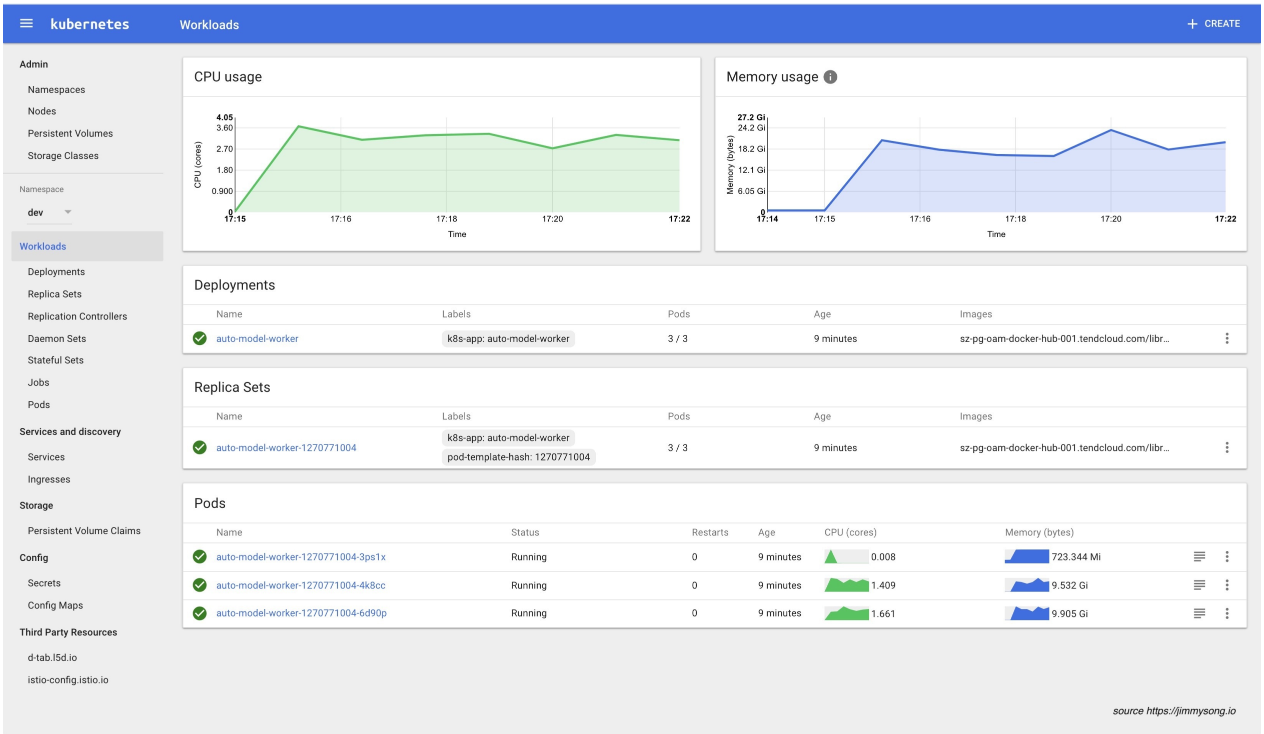
仍然使用 `spark-submit` 提交spark任务，可以直接指定kubernetes API server地址，下面的命令提交本地jar包到kubernetes集群上运行，同时指定了运行任务的用户、提交命名的用户、运行的excutor实例数、driver和executor的资源限制、使用的spark版本等信息。

详细使用说明见Apache Spark on Kubernetes用户指南 - jimmysong.io。

```
./spark-submit \
  --deploy-mode cluster \
  --class com.talkingdata.alluxio.hadooptest \
  --master k8s://https://172.20.0.113:6443 \
  --kubernetes-namespace spark-cluster \
  --conf spark.kubernetes.driverEnv.SPARK_USER=hadoop \
  --conf spark.kubernetes.driverEnv.HADOOP_USER_NAME=hadoop \
  --conf spark.executorEnv.HADOOP_USER_NAME=hadoop \
  --conf spark.executorEnv.SPARK_USER=hadoop \
  --conf spark.kubernetes.authenticate.driver.serviceAccountName=
spark \
  --conf spark.driver.memory=100G \
  --conf spark.executor.memory=10G \
  --conf spark.driver.cores=30 \
  --conf spark.executor.cores=2 \
  --conf spark.driver.maxResultSize=10240m \
  --conf spark.kubernetes.driver.limit.cores=32 \
  --conf spark.kubernetes.executor.limit.cores=3 \
  --conf spark.kubernetes.executor.memoryOverhead=2g \
  --conf spark.executor.instances=5 \
  --conf spark.app.name=spark-pi \
  --conf spark.kubernetes.driver.docker.image=harbor-001.jimmyso
ng.io/library/spark-driver:v2.1.0-kubernetes-0.3.1-1 \
  --conf spark.kubernetes.executor.docker.image=harbor-001.jimmy
song.io/library/spark-executor:v2.1.0-kubernetes-0.3.1-1 \
  --conf spark.kubernetes.initcontainer.docker.image=harbor-001.
jimmysong.io/library/spark-init:v2.1.0-kubernetes-0.3.1-1 \
  --conf spark.kubernetes.resourceStagingServer.uri=http://172.2
0.0.114:31000 \
~/Downloads/tendcloud_2.10-1.0.jar
```

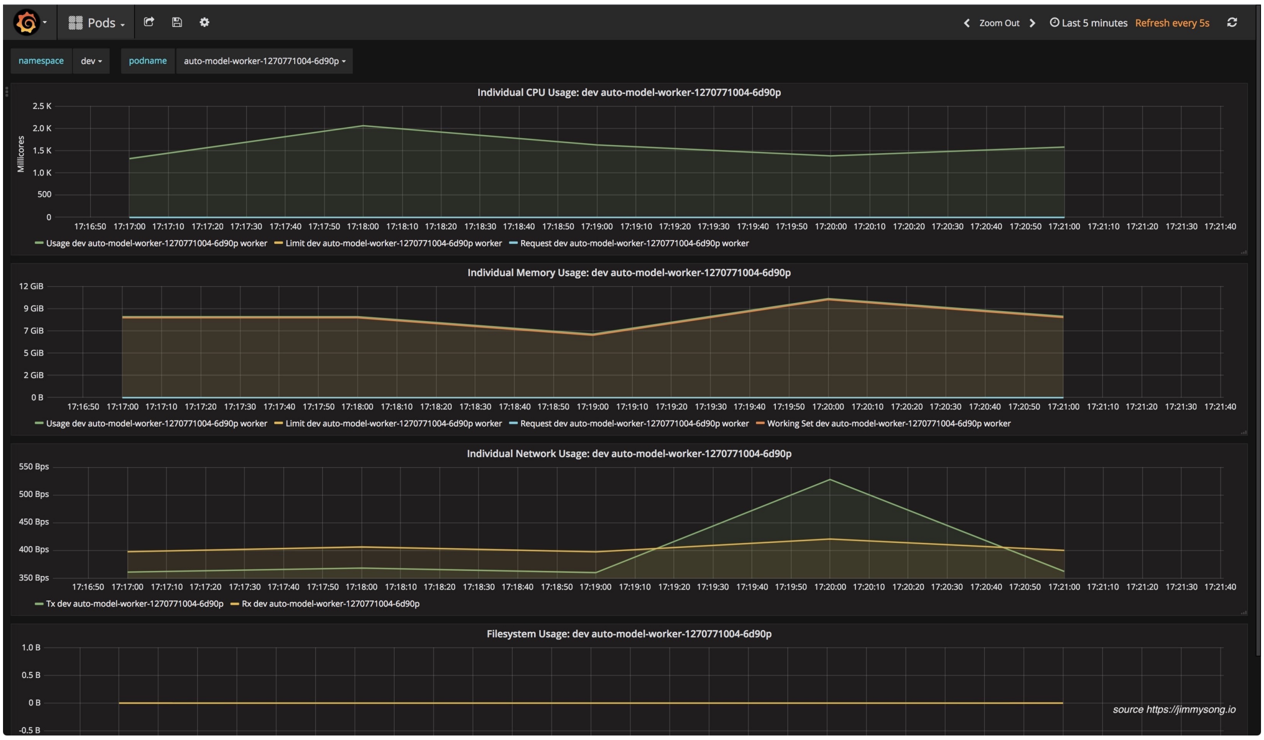
监控

下图是从Kubernetes dashboard上看到的spark-cluster这个namespace上运行的应用情况。



图片 - *dashboard*

下图是从Grafana监控页面上查看到的某个executor资源占用情况。



图片 - *Grafana*

参考

- 迁移到云原生应用架构指南
- Cloud Native Go - 已由电子工业出版社出版
- Cloud Native Python - 将由电子工业出版社出版
- Istio Service Mesh 中文文档

云原生应用之路——从Kubernetes到Cloud Native

从**Kubernetes**到**Cloud Native——**云原生应用之路，这是我最近在ArchSummit2017北京站 和 数人云&TalkingData合办的Service Mesh is coming meetup 中分享的话题。

本文简要介绍了容器技术发展的路径，为何Kubernetes的出现是容器技术发展到这一步的必然选择，而为何Kuberentes又将成为云原生应用的基石。

我的分享按照这样的主线展开：容器->Kubernetes->微服务->Cloud Native（云原生）->Service Mesh（服务网格）->使用场景->Open Source（开源）。

容器

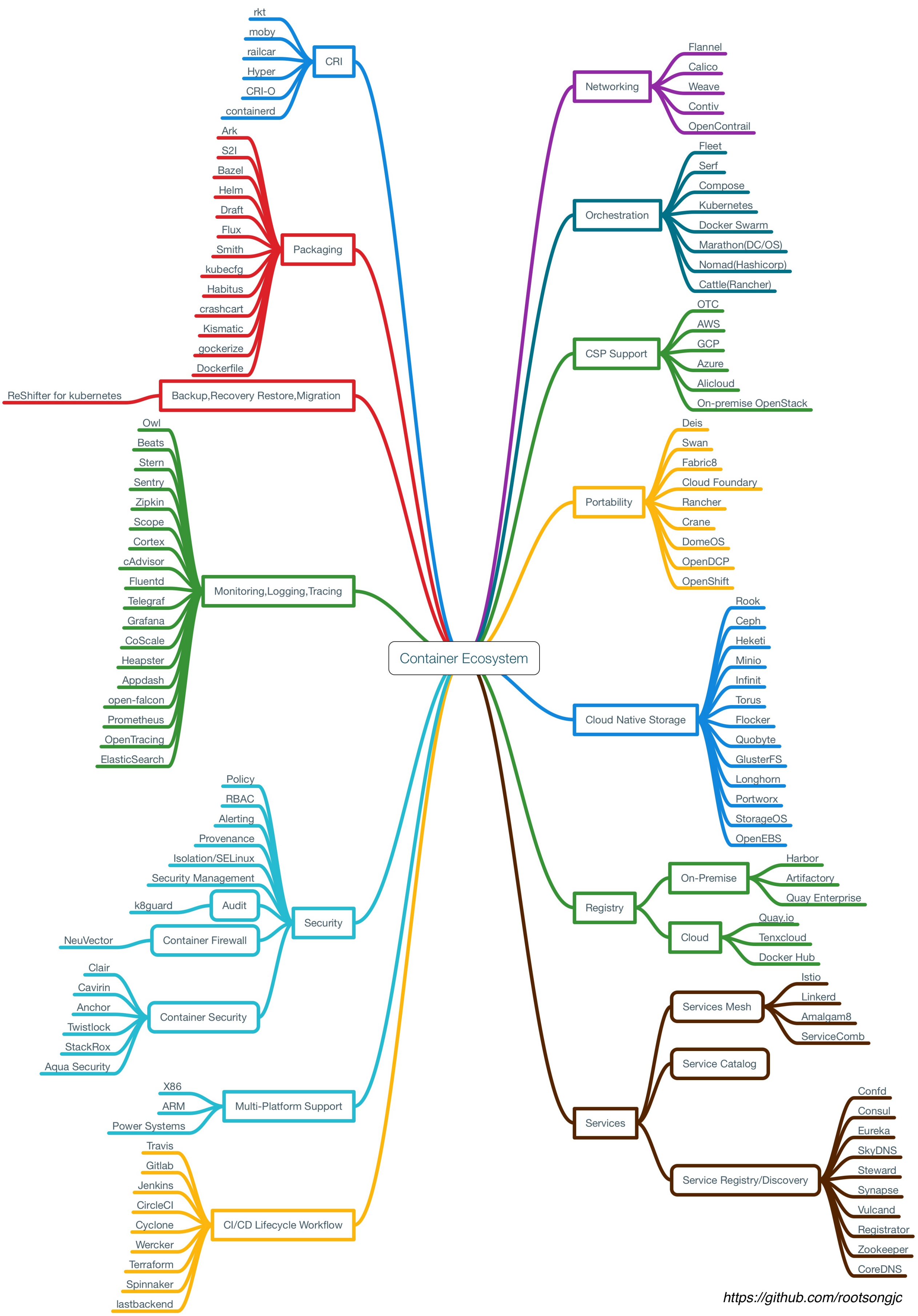
> 容器——Cloud Native的基石



图片 - *Cloud Native*容器实验室

容器最初是通过开发者工具而流行，可以使用它来做隔离的开发测试环境和持续集成环境，这些都是因为容器轻量级，易于配置和使用带来的优势，docker和docker-compose这样的工具极大的方便的了应用开发环境的搭建，开发者就像是化学家一样在其中小心翼翼的进行各种调试和开发。

随着容器的在开发者中的普及，已经大家对CI流程的熟悉，容器周边的各种工具蓬勃发展，俨然形成了一个小生态，在2016年达到顶峰，下面这张是我画的容器生态图：



图片 - 容器生态图 *Container ecosystem*

该生态涵盖了容器应用中从镜像仓库、服务编排、安全管理、持续集成与发布、存储和网络管理等各个方面，随着在单主机中运行容器的成熟，集群管理和容器编排成为容器技术亟待解决的问题。譬如化学家在实验室中研究出来的新产品，如何推向市场，进行大规模生产，成了新的议题。

为什么使用Kubernetes

> Kubernetes——让容器应用进入大规模工业生产。

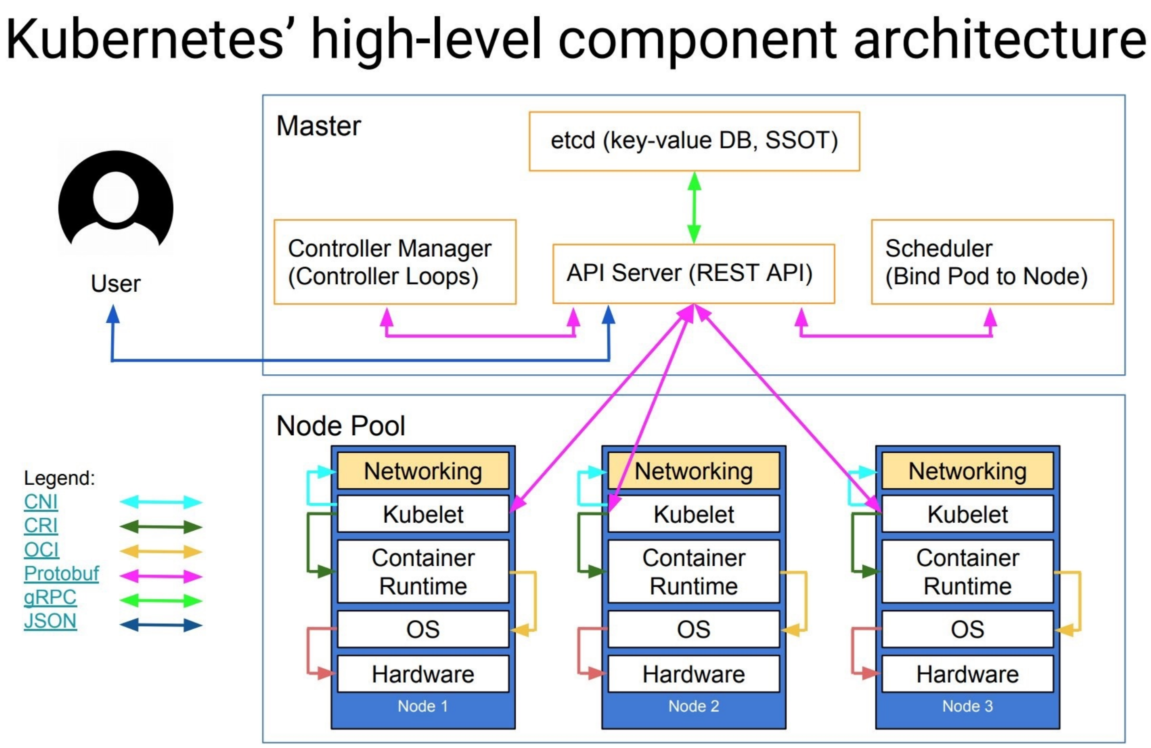


图片 - *Cloud Native*油井

Kubernetes是容器编排系统的事实标准

在单机上运行容器，无法发挥它的最大效能，只有形成集群，才能最大程度发挥容器的良好隔离、资源分配与编排管理的优势，而对于容器的编排管理，Swarm、Mesos和Kubernetes的大战已经基本宣告结束，kubernetes成为了无可争议的赢家。

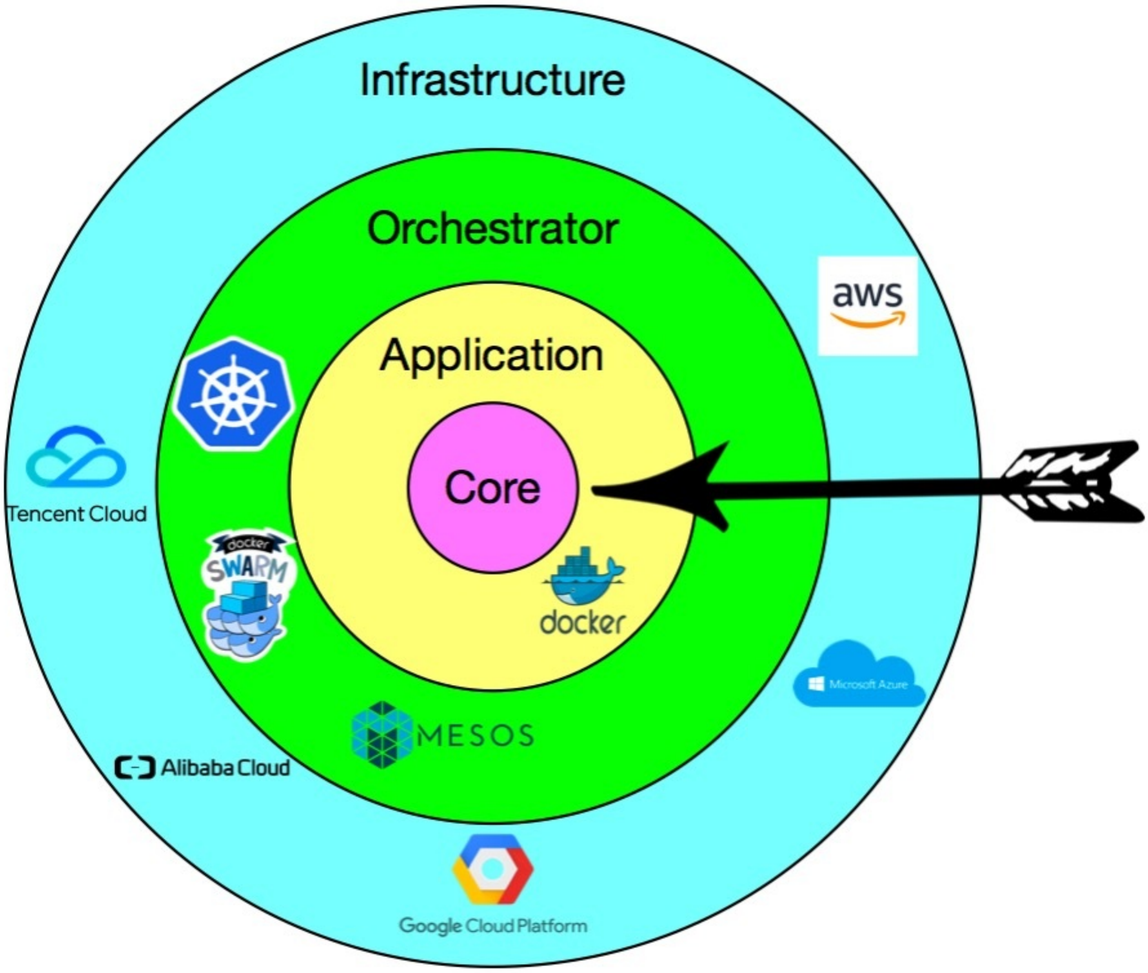
下面这张图是Kubernetes的架构图（图片来自网络），其中显示了组件之间交互的接口CNI、CRI、OCI等，这些将Kubernetes与某款具体产品解耦，给用户最大的定制程度，使得Kubernetes有机会成为跨云的真正的云原生应用的操作系统。



图片 - *Kuberentes*架构

随着Kubernetes的日趋成熟，“Kubernetes is becoming boring”，基于该“操作系统”之上构建的适用于不同场景的应用将成为新的发展方向，就像我们将石油开采出来后，提炼出汽油、柴油、沥青等等，所有的材料都将找到自己的用途，Kubernetes也是，毕竟我们谁也不是为了部署和管理容器而用Kubernetes，承载其上的应用才是价值之所在。

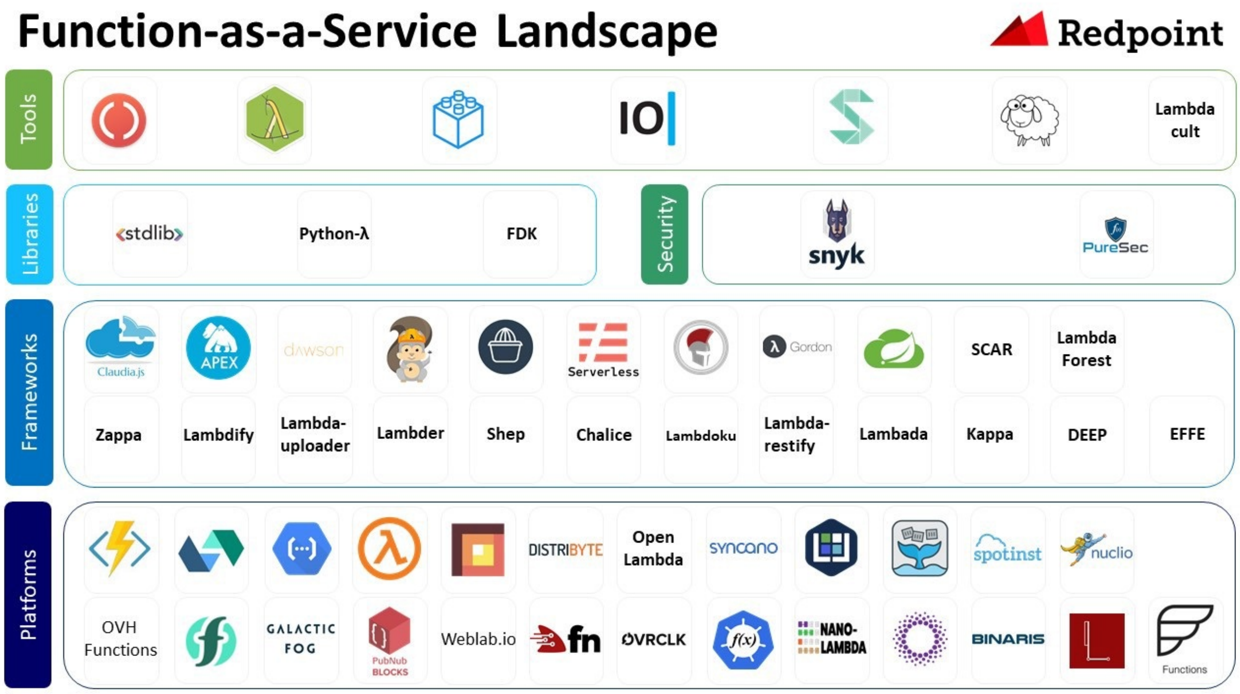
云原生的核心目标



图片 - *Cloud Native Core target*

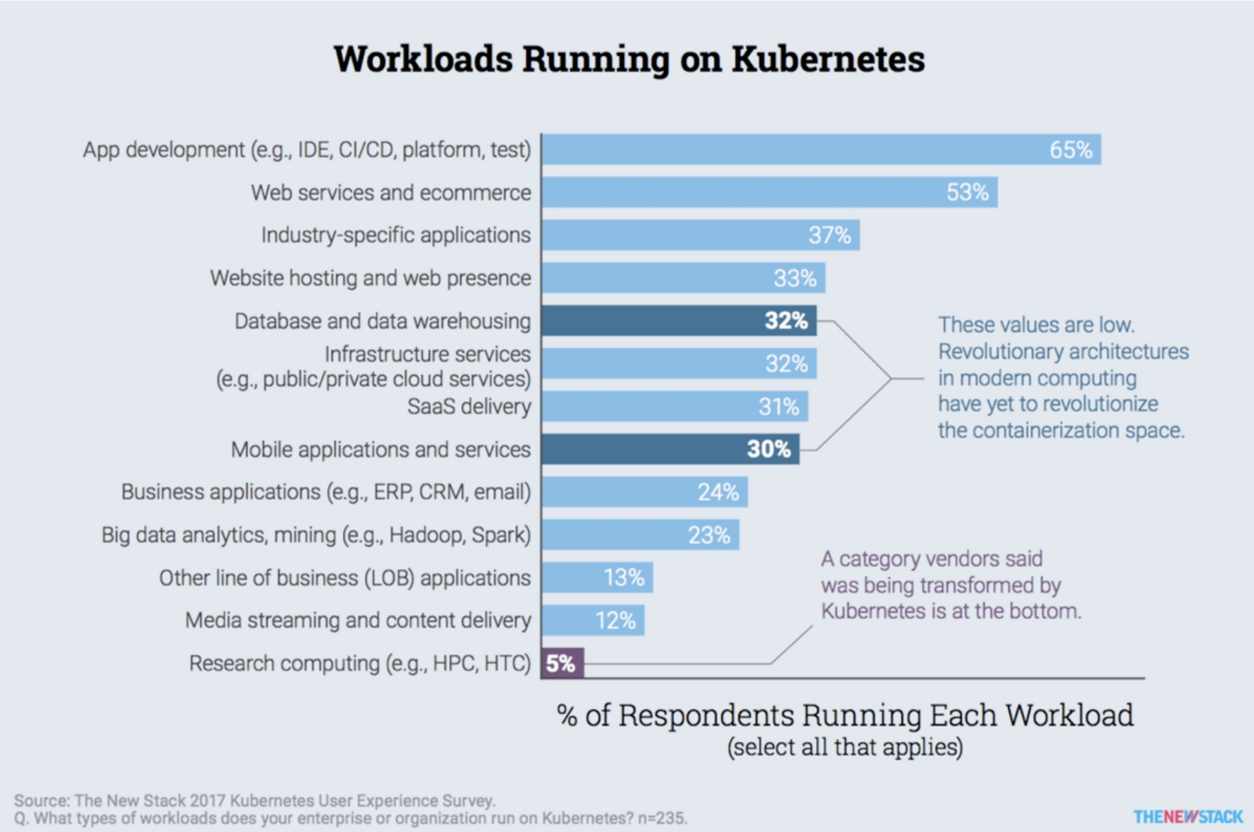
云已经可以为我们提供稳定可以唾手可得的基础设施，但是业务上云成了一个难题，Kubernetes的出现与其说是从最初的容器编排解决方案，倒不如说是为了解决应用上云（即云原生应用）这个难题。

包括微服务和FaaS/Serverless架构，都可以作为云原生应用的架构。



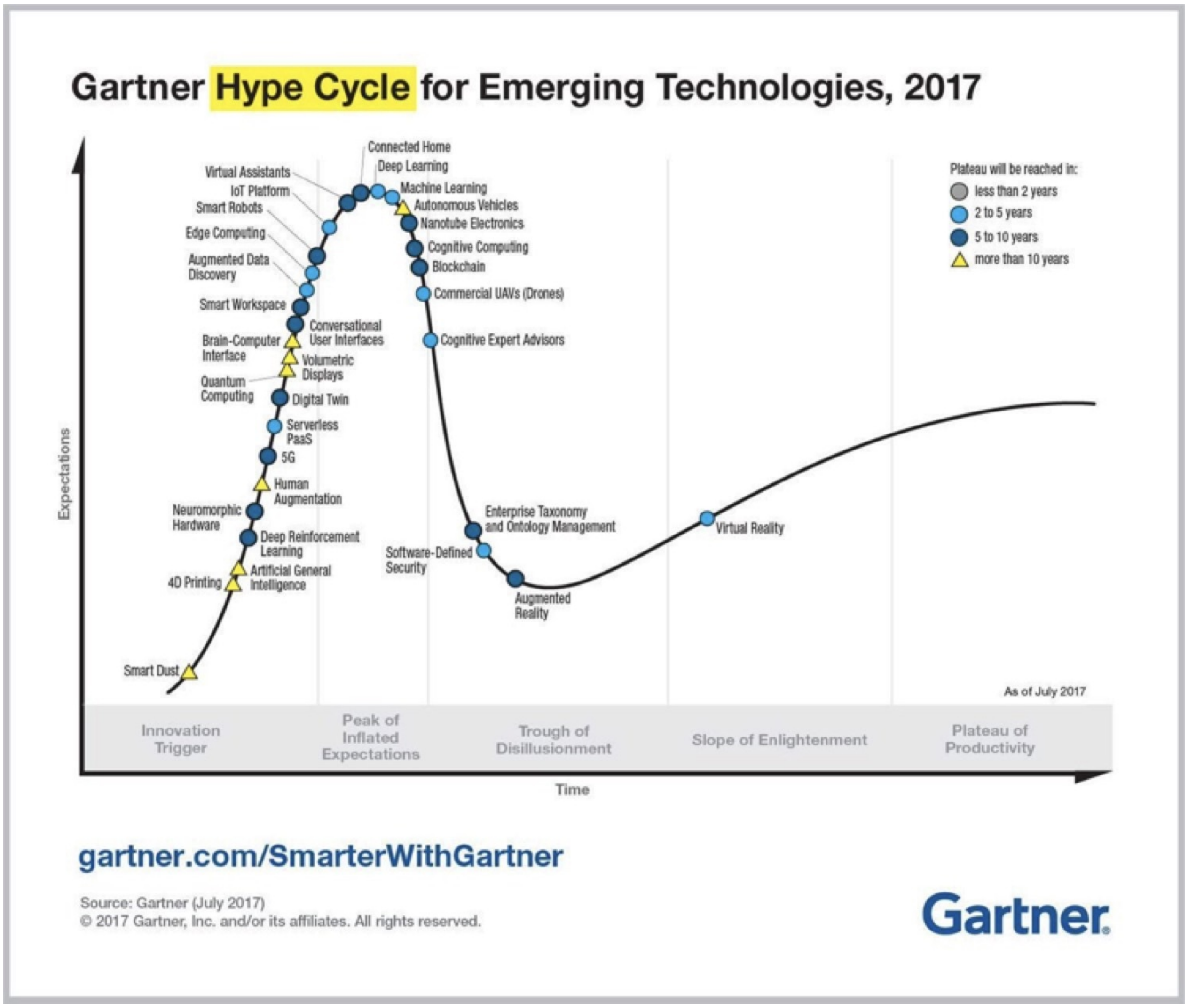
图片 - *FaaS Landscape*

但就2017年为止，kubernetes的主要使用场景也主要作为应用开发测试环境、CI/CD和运行Web应用这几个领域，如下图TheNewStack的Kubernetes生态状况调查报告所示。



图片 - *Workloads running on Kubernetes*

另外基于Kubernetes的构建PaaS平台和Serverless也处于爆发的准备的阶段，如下图中Gartner的报告中所示：



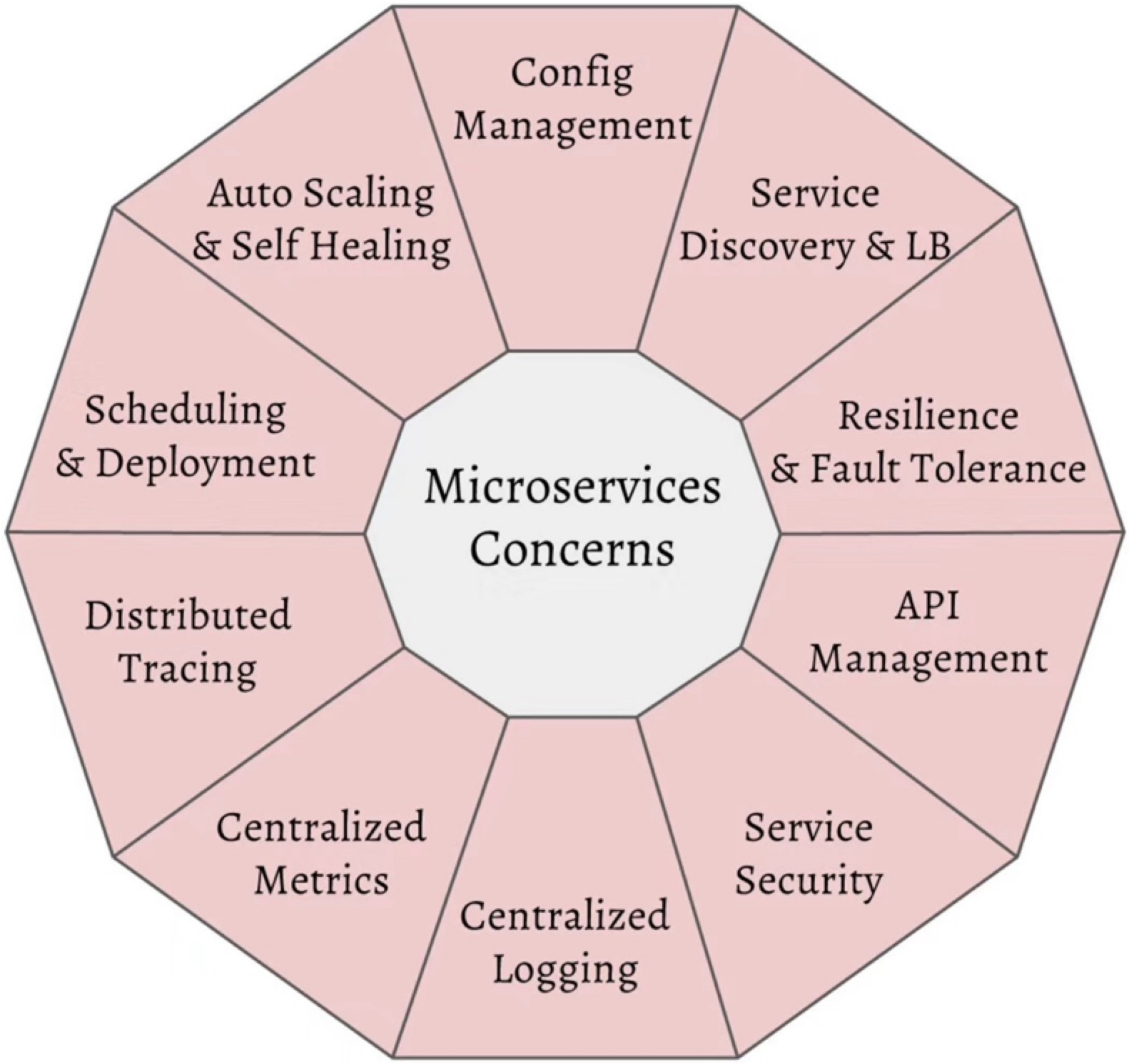
图片 - *Gartner技术爆发趋势图2017*

当前各大公有云如Google GKE、微软Azure ACS、亚马逊EKS（2018年上线）、VmWare、Pivotal、腾讯云、阿里云等都提供了Kuberentes服务。

微服务

> 微服务——Cloud Native的应用架构。

下图是Bilgin Ibryam给出的微服务中应该关心的主题，图片来自RedHat Developers。

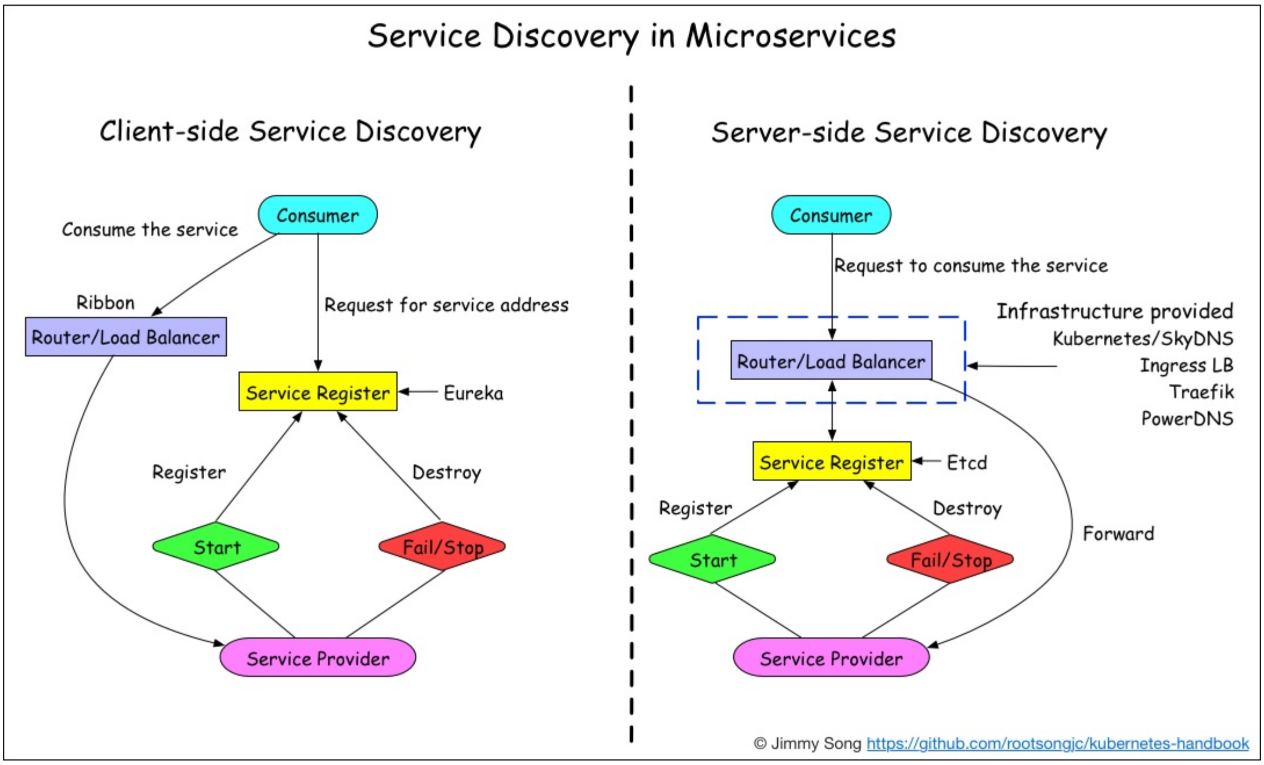


图片 - *Microservices concerns*

微服务带给我们很多开发和部署上的灵活性和技术多样性，但是也增加了服务调用的开销、分布式系统管理、调试与服务治理方面的难题。

当前最成熟最完整的微服务框架可以说非Spring莫属，而Spring又仅限于Java语言开发，其架构本身又跟Kubernetes存在很多重合的部分，如何探索将Kubernetes作为微服务架构平台就成为一个热点话题。

就拿微服务中最基础的服务注册发现功能来说，其方式分为客户端服务发现和服务端服务发现两种，Java应用中常用的方式是使用Eureka和Ribbon做服务注册发现和负载均衡，这属于客户端服务发现，而在Kubernetes中则可以使用DNS、Service和Ingress来实现，不需要修改应用代码，直接从网络层面来实现。



图片 - 两种服务发现方式

Cloud Native

> DevOps——通向云原生的云梯

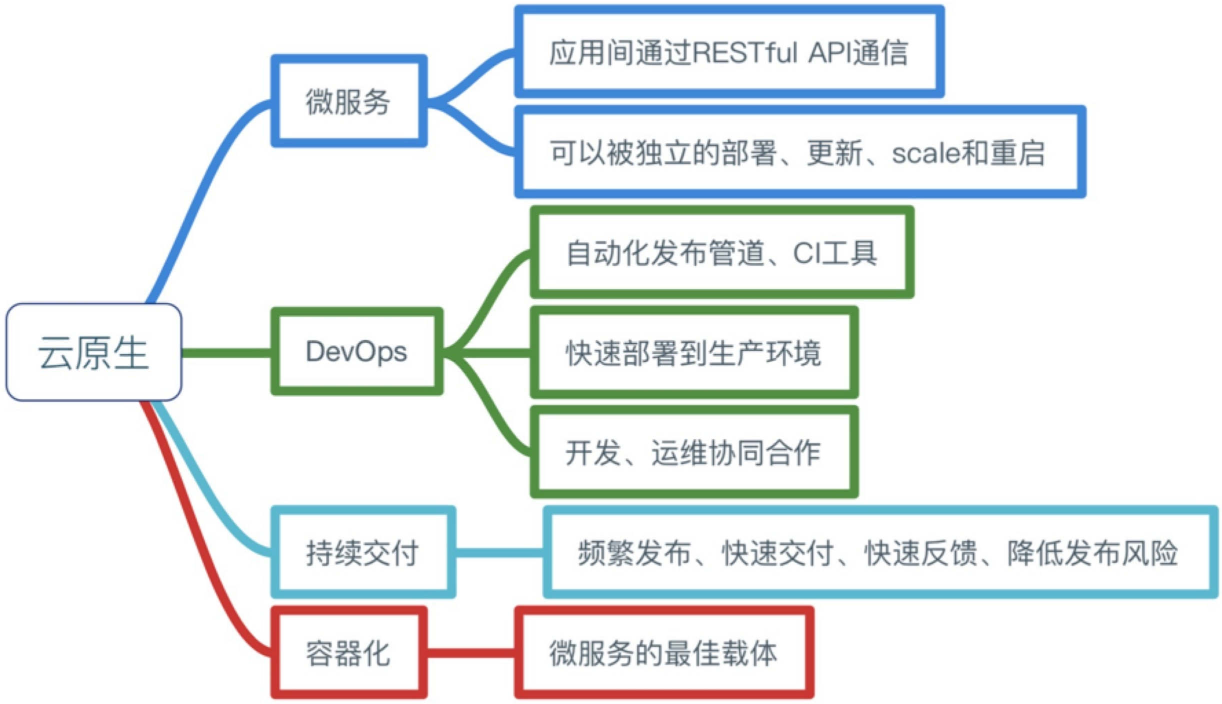


图片 - *Cloud Native Pipeline*

CNCF（云原生计算基金会）给出了云原生应用的三大特征：

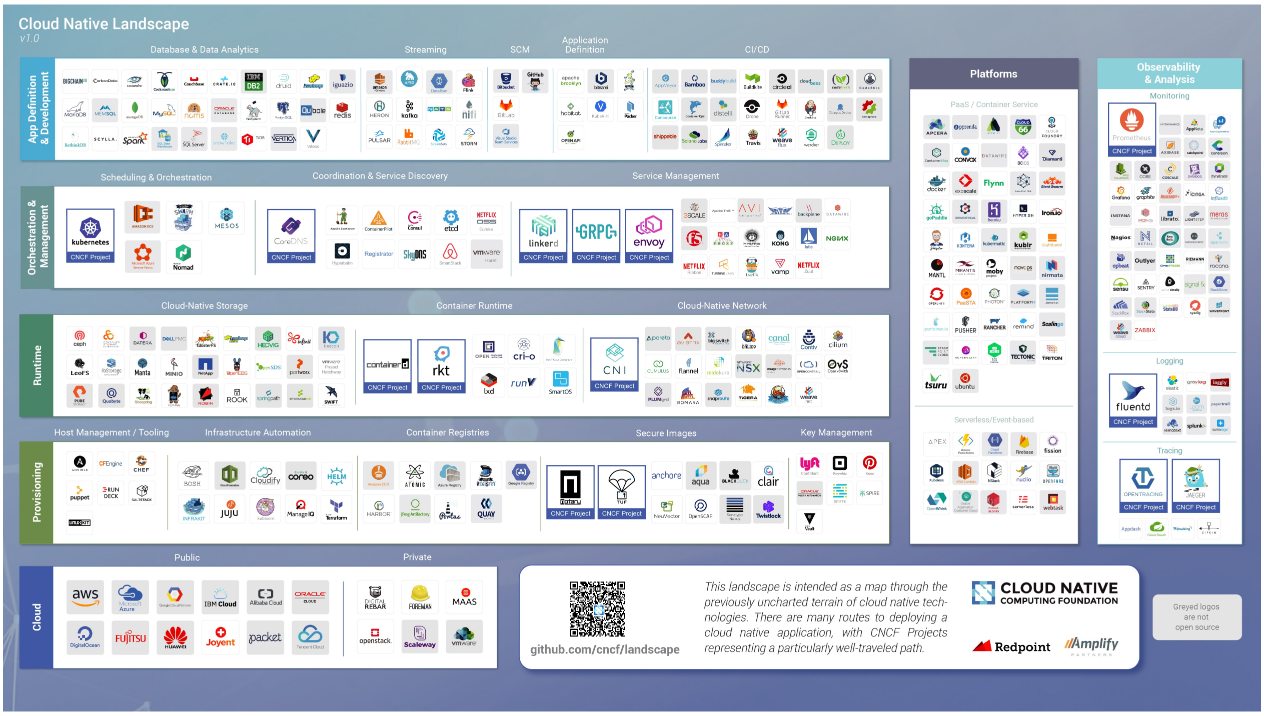
- 容器化包装：软件应用的进程应该包装在容器中独立运行。
- 动态管理：通过集中式的编排调度系统来动态的管理和调度。
- 微服务化：明确服务间的依赖，互相解耦。

下图是我整理的关于云原生所需要的能力和特征。



图片 - *Cloud Native Features*

CNCF所托管的应用（目前已达12个），即朝着这个目标发展，其公布的Cloud Native Landscape，给出了云原生生态的参考体系。



图片 - *Cloud Native Landscape v1.0*

使用**Kubernetes**构建云原生应用

我们都是知道Heroku推出了适用于PaaS的12 factor app的规范，包括如下要素：

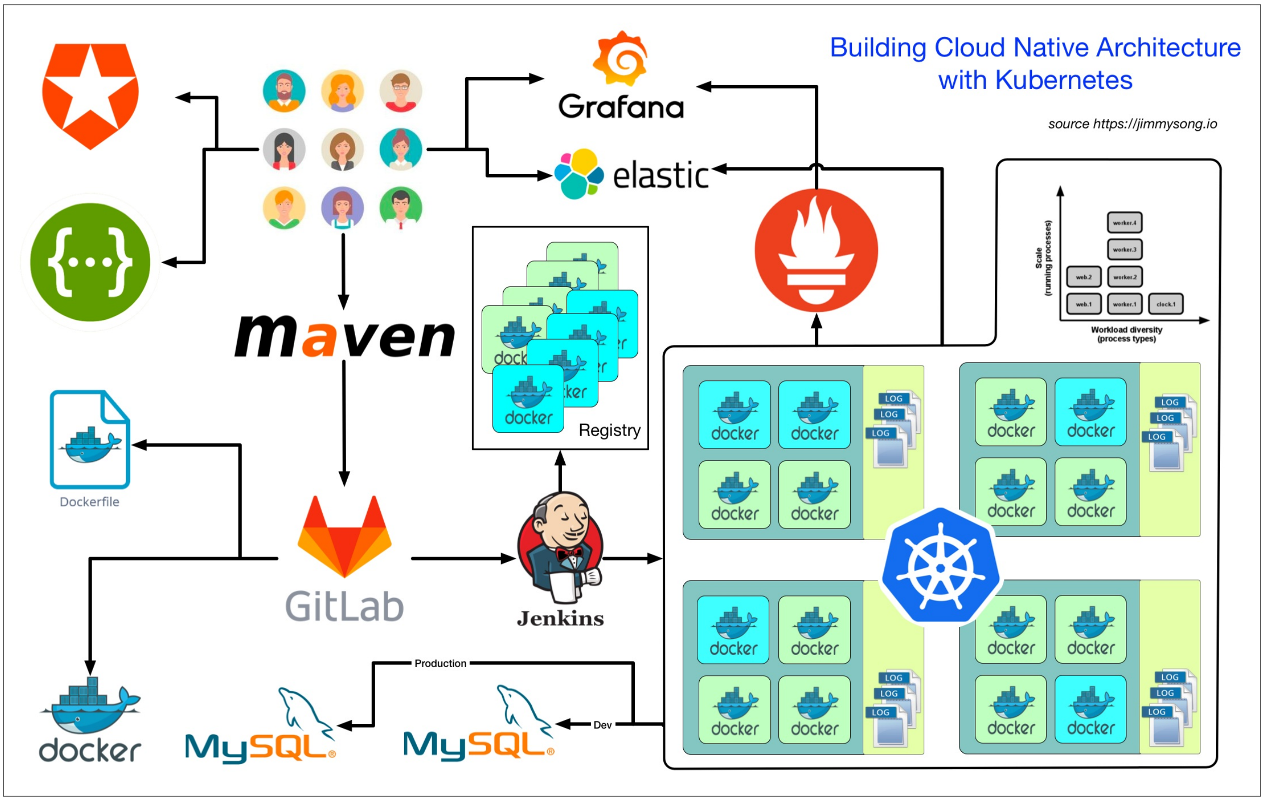
1. 基准代码
2. 依赖管理
3. 配置
4. 后端服务
5. 构建，发布，运行
6. 无状态进程
7. 端口绑定
8. 并发
9. 易处理
10. 开发环境与线上环境等价
11. 日志作为事件流
12. 管理进程

另外还有补充的三点：

- API声明管理
- 认证和授权

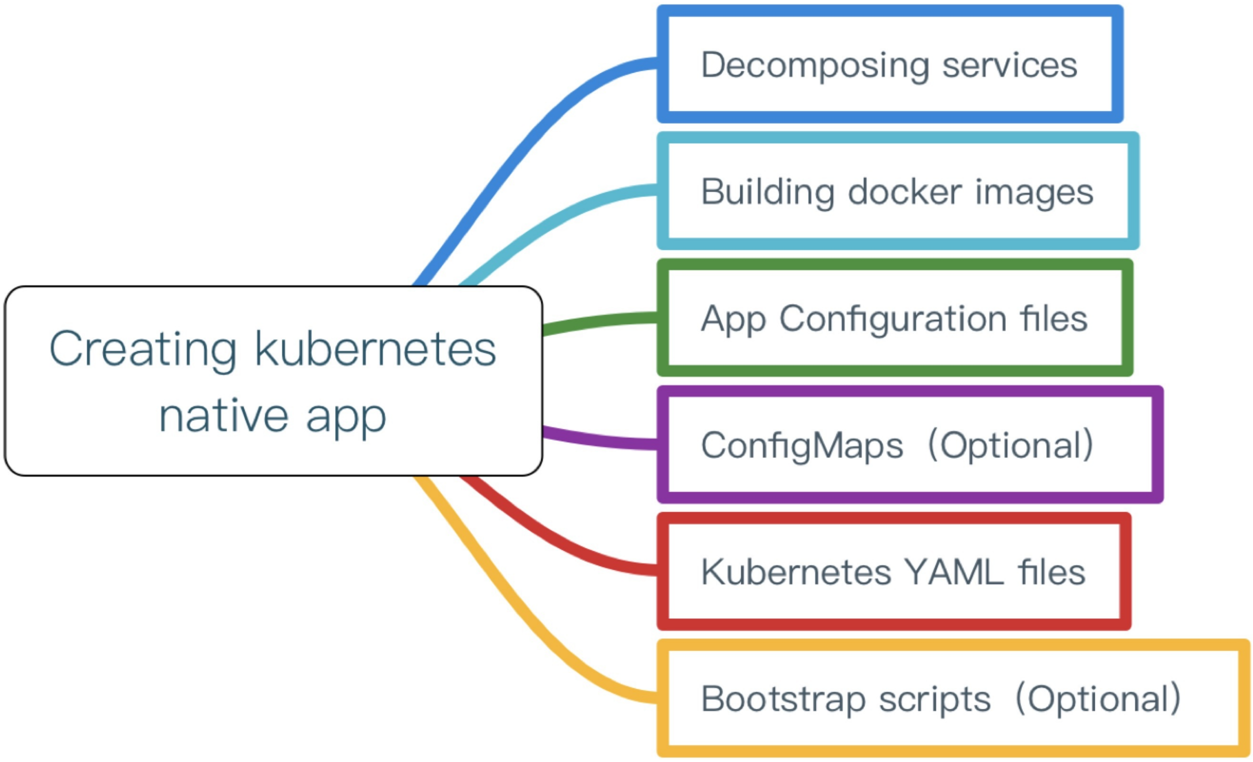
- 监控与告警

如果落实的具体的工具，请看下图，使用Kubernetes构建云原生架构：



图片 - *Building a Cloud Native Architecture with Kubernetes followed 12 factor app*

结合这12因素对开发或者改造后的应用适合部署到Kubernetes之上，基本流程如下图所示：



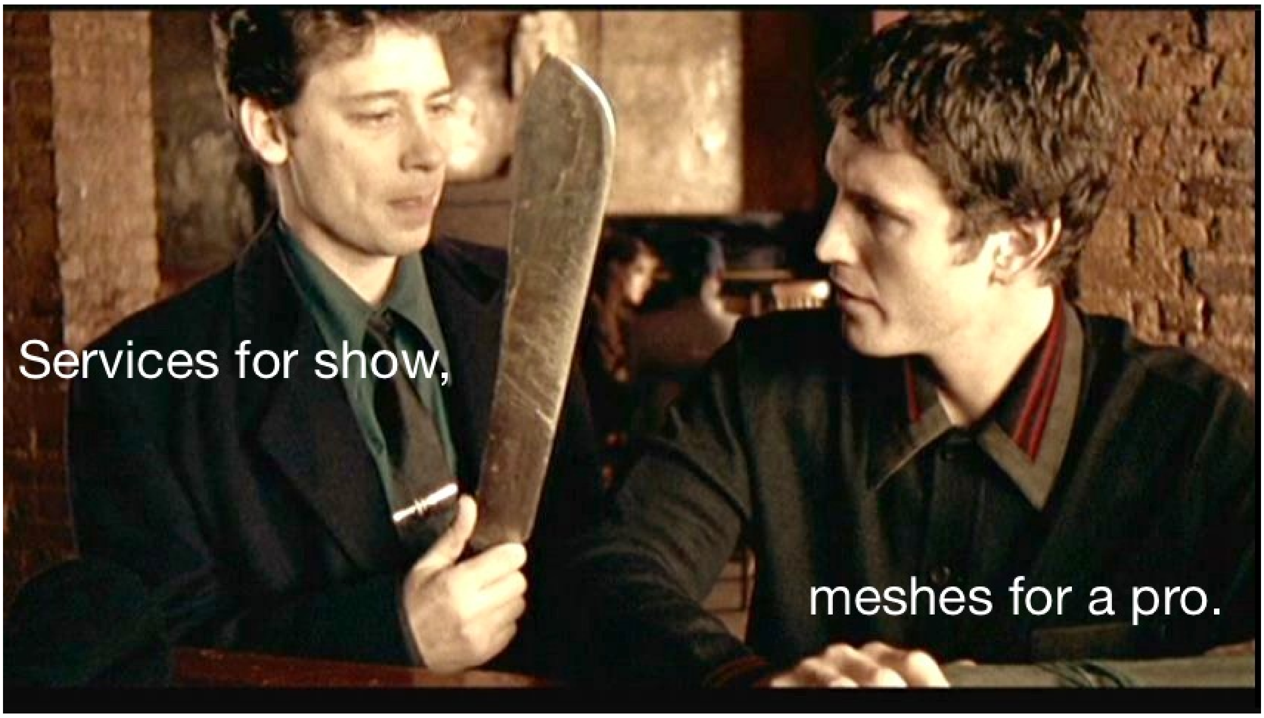
图片 - *Creating Kubernetes native app*

迁移到云架构

迁移到云端架构，相对单体架构来说会带来很多挑战。比如自动的持续集成与发布、服务监控的变革、服务暴露、权限的管控等。这些具体细节请参考**Kubernetes-handbook**中的说明：https://jimmysong.io/kubernetes-handbook，在此就不细节展开，另外推荐一本我翻译的由Pivotal出品的电子书——Migrating to Cloud Native Application Architectures，地址：https://jimmysong.io/migrating-to-cloud-native-application-architectures/。

Service Mesh

> Services for show, meshes for a pro.



图片 - *Service Mesh*中国社区*slogan*

Kubernetes中的应用将作为微服务运行，但是Kuberentes本身并没有给出微服务治理的解决方案，比如服务的限流、熔断、良好的灰度发布支持等。

Service mesh可以用来做什么

- Traffic Management：API网关
- Observability：服务调用和性能分析
- Policy Enforcment：控制服务访问策略
- Service Identity and Security：安全保护

Service mesh的特点

- 专用的基础设施层
- 轻量级高性能网络代理
- 提供安全的、快速的、可靠地服务间通讯
- 扩展kubernetes的应用负载均衡机制，实现灰度发布
- 完全解耦于应用，应用可以无感知，加速应用的微服务和云原生转型

使用Service Mesh将可以有效的治理Kuberentes中运行的服务，当前开源的Service Mesh有：

- Linkderd：https://linkerd.io，由最早提出Service Mesh的公司Buoyant开源，

创始人来自Twitter
- Envoy：https://www.envoyproxy.io/，Lyft开源的，可以在Istio中使用Sidecar模式运行
- Istio：https://istio.io，由Google、IBM、Lyft联合开发并开源
- Conduit：https://conduit.io，同样由Buoyant开源的轻量级的基于Kubernetes的Service Mesh

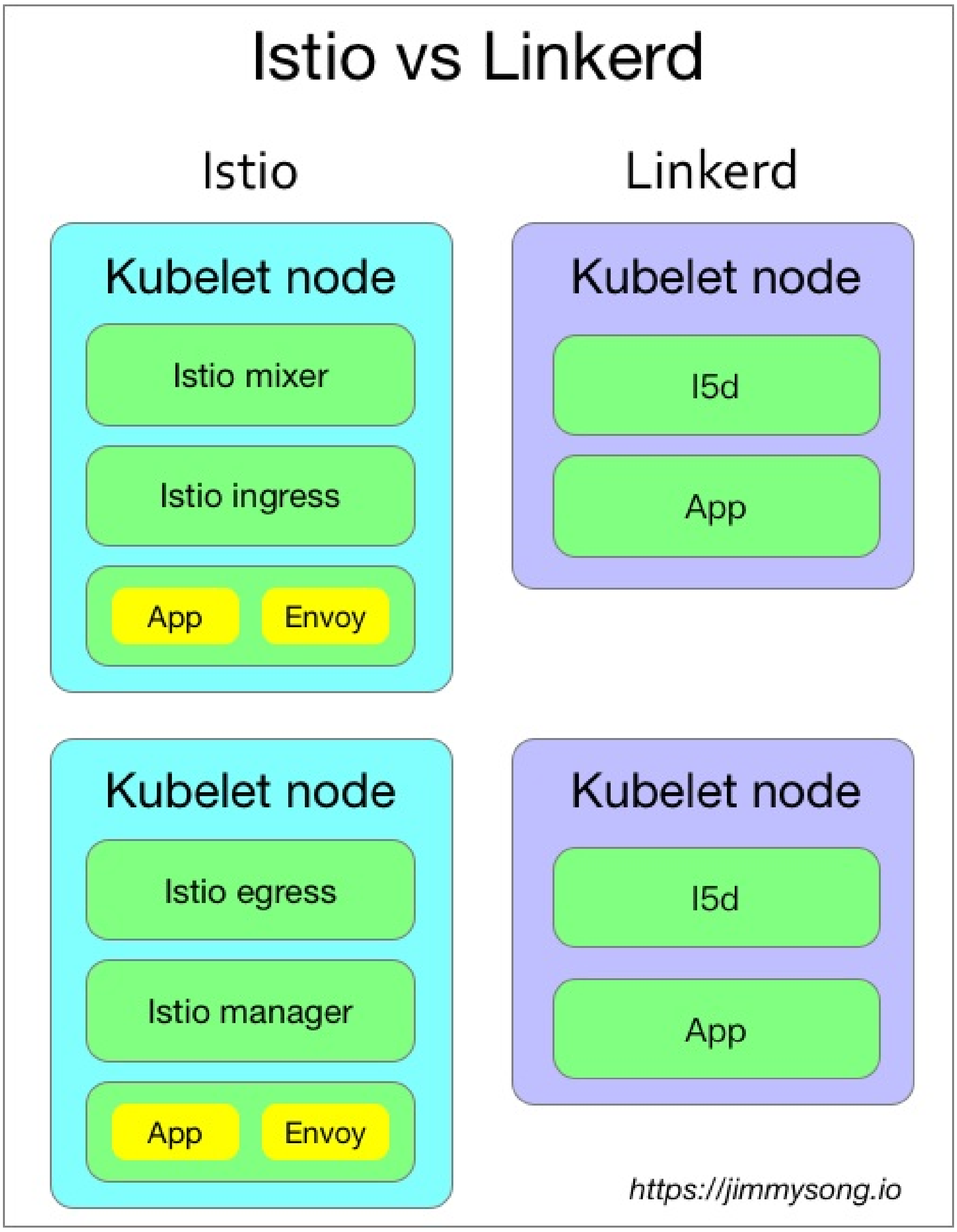
此外还有很多其它的Service Mesh鱼贯而出，请参考awesome-cloud-native。

Istio VS Linkerd

Linkerd和Istio是最早开源的Service Mesh，它们都支持Kubernetes，下面是它们之间的一些特性对比。

Feature	Istio	Linkerd
部署架构	Envoy/Sidecar	DaemonSets
易用性	复杂	简单
支持平台	kuberentes	kubernetes/mesos/Istio/local
当前版本	0.3.0	1.3.3
是否已有生产部署	否	是

关于两者的架构可以参考各自的官方文档，我只从其在kubernetes上的部署结构来说明其区别。



图片 - *istio vs linkerd*

Istio的组件复杂，可以分别部署的kubernetes集群中，但是作为核心路由组件**Envoy**是以**Sidecar**形式与应用运行在同一个Pod中的，所有进入该Pod中的流量都需要先经过Envoy。

Linker的部署十分简单，本身就是一个镜像，使用Kubernetes的DaemonSet方式在每个node节点上运行。

更多信息请参考kubernetes-handbook。

使用场景

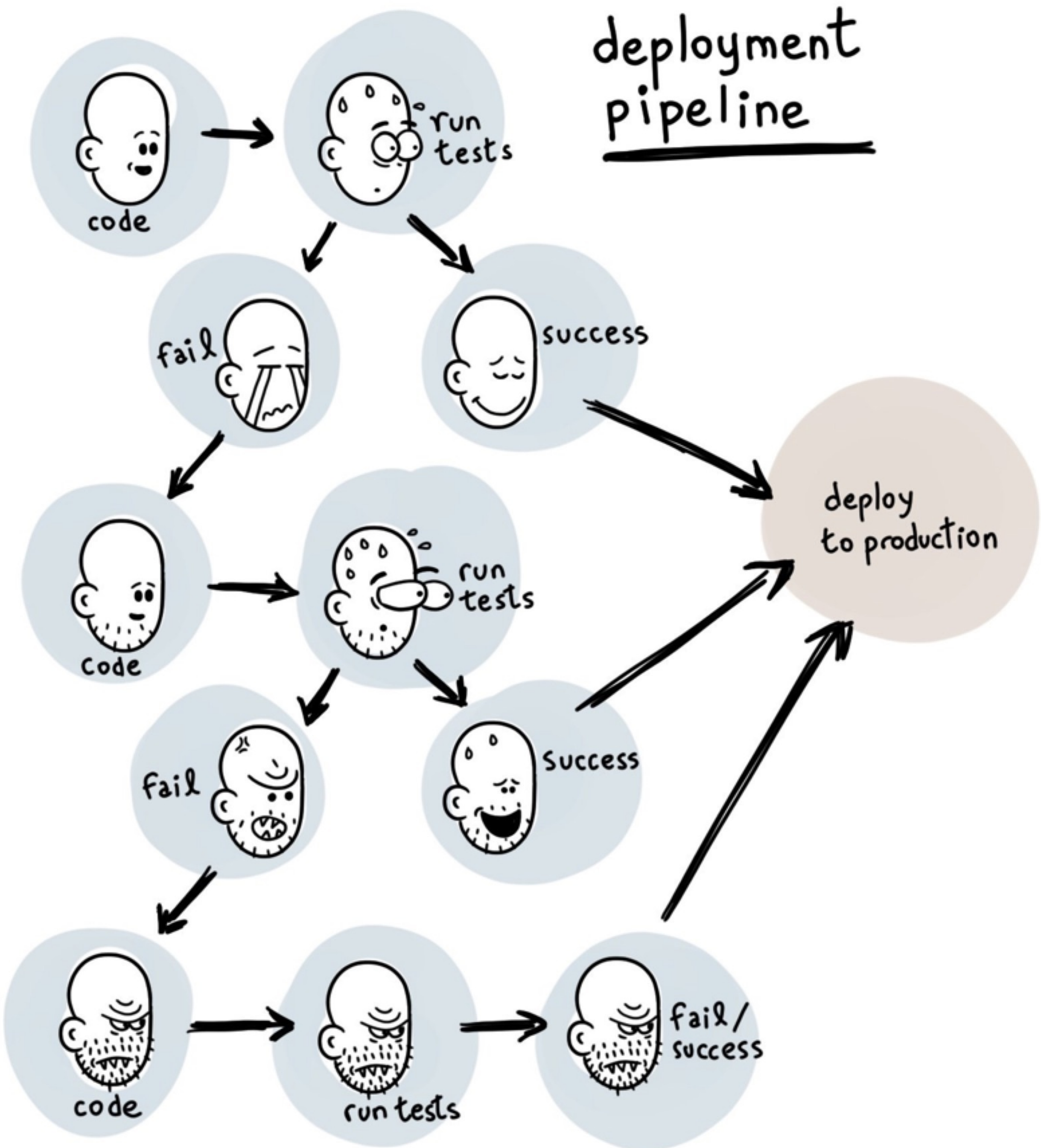
> Cloud Native的大规模工业生产



图片 - *Cloud Native factory*

GitOps

给开发者带来最大配置和上线的灵活性，践行DevOps流程，改善研发效率，下图这样的情况将更少发生。



图片 - *Deployment pipeline*

我们知道Kubernetes中的所有应用的部署都是基于YAML文件的，这实际上就是一种**Infrastructure as code**，完全可以通过Git来管控基础设施和部署环境的变更。

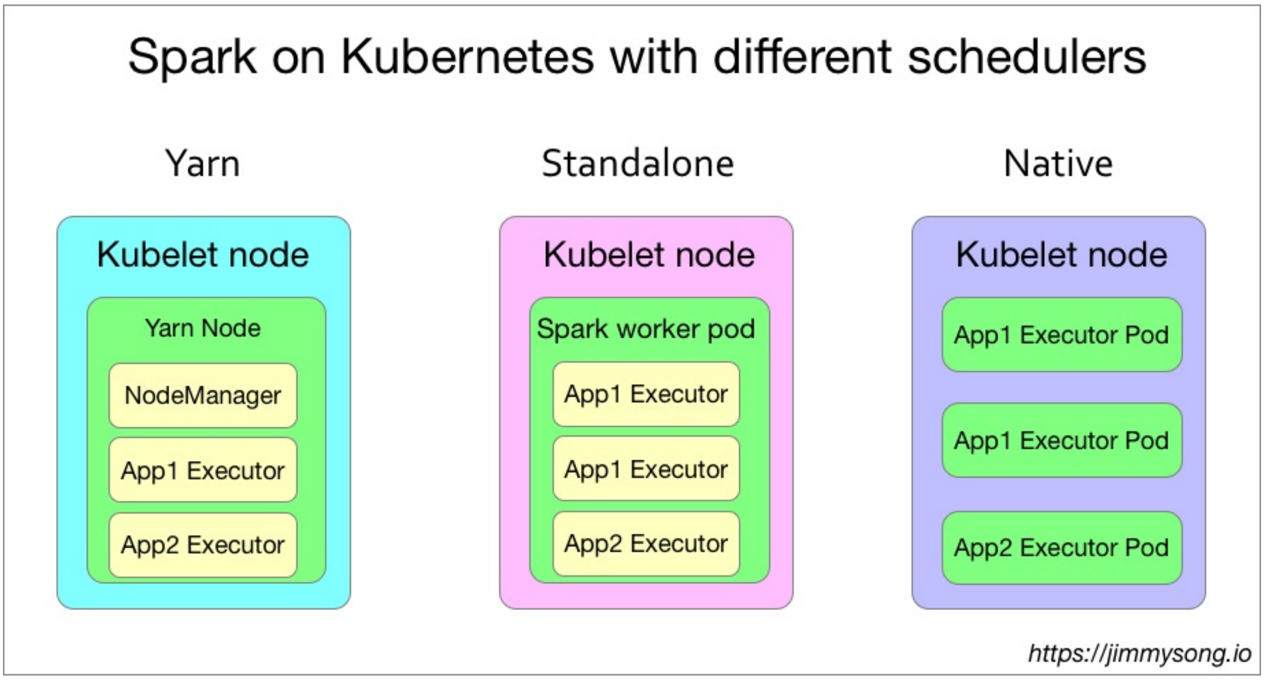
Big Data

Spark现在已经非官方支持了基于Kuberentes的原生调度，其具有以下特点：

- Kubernetes原生调度：与yarn、mesos同级
- 资源隔离，粒度更细：以namespace来划分用户
- 监控的变革：单次任务资源计量
- 日志的变革：pod的日志收集

Feature	Yarn	Kubernetes
queue	queue	namespace
instance	ExcutorContainer	Executor Pod
network	host	plugin
heterogeneous	no	yes
security	RBAC	ACL

下图是在Kubernetes上运行三种调度方式的spark的单个节点的应用部分对比：



图片 - *Spark on Kubernetes with different schedulers*

从上图中可以看到在Kubernetes上使用YARN调度、standalone调度和kubernetes原生调度的方式，每个node节点上的Pod内的spark Executor分布，毫无疑问，使用kubernetes原生调度的spark任务才是最节省资源的。

提交任务的语句看起来会像是这样的：

```
./spark-submit \
  --deploy-mode cluster \
  --class com.talkingdata.alluxio.hadooptest \
  --master k8s://https://172.20.0.113:6443 \
  --kubernetes-namespace spark-cluster \
  --conf spark.kubernetes.driverEnv.SPARK_USER=hadoop \
  --conf spark.kubernetes.driverEnv.HADOOP_USER_NAME=hadoop \
  --conf spark.executorEnv.HADOOP_USER_NAME=hadoop \
  --conf spark.executorEnv.SPARK_USER=hadoop \
  --conf spark.kubernetes.authenticate.driver.serviceAccountName=
spark \
  --conf spark.driver.memory=100G \
  --conf spark.executor.memory=10G \
  --conf spark.driver.cores=30 \
  --conf spark.executor.cores=2 \
  --conf spark.driver.maxResultSize=10240m \
  --conf spark.kubernetes.driver.limit.cores=32 \
  --conf spark.kubernetes.executor.limit.cores=3 \
  --conf spark.kubernetes.executor.memoryOverhead=2g \
  --conf spark.executor.instances=5 \
  --conf spark.app.name=spark-pi \
  --conf spark.kubernetes.driver.docker.image=spark-driver:v2.1.
0-kubernetes-0.3.1-1 \
  --conf spark.kubernetes.executor.docker.image=spark-executor:v
2.1.0-kubernetes-0.3.1-1 \
  --conf spark.kubernetes.initcontainer.docker.image=spark-init:
v2.1.0-kubernetes-0.3.1-1 \
  --conf spark.kubernetes.resourceStagingServer.uri=http://172.2
0.0.114:31000 \
~/Downloads/tendcloud_2.10-1.0.jar
```

关于支持Kubernetes原生调度的Spark请参考：https://jimmysong.io/spark-on-k8s/

Open Source

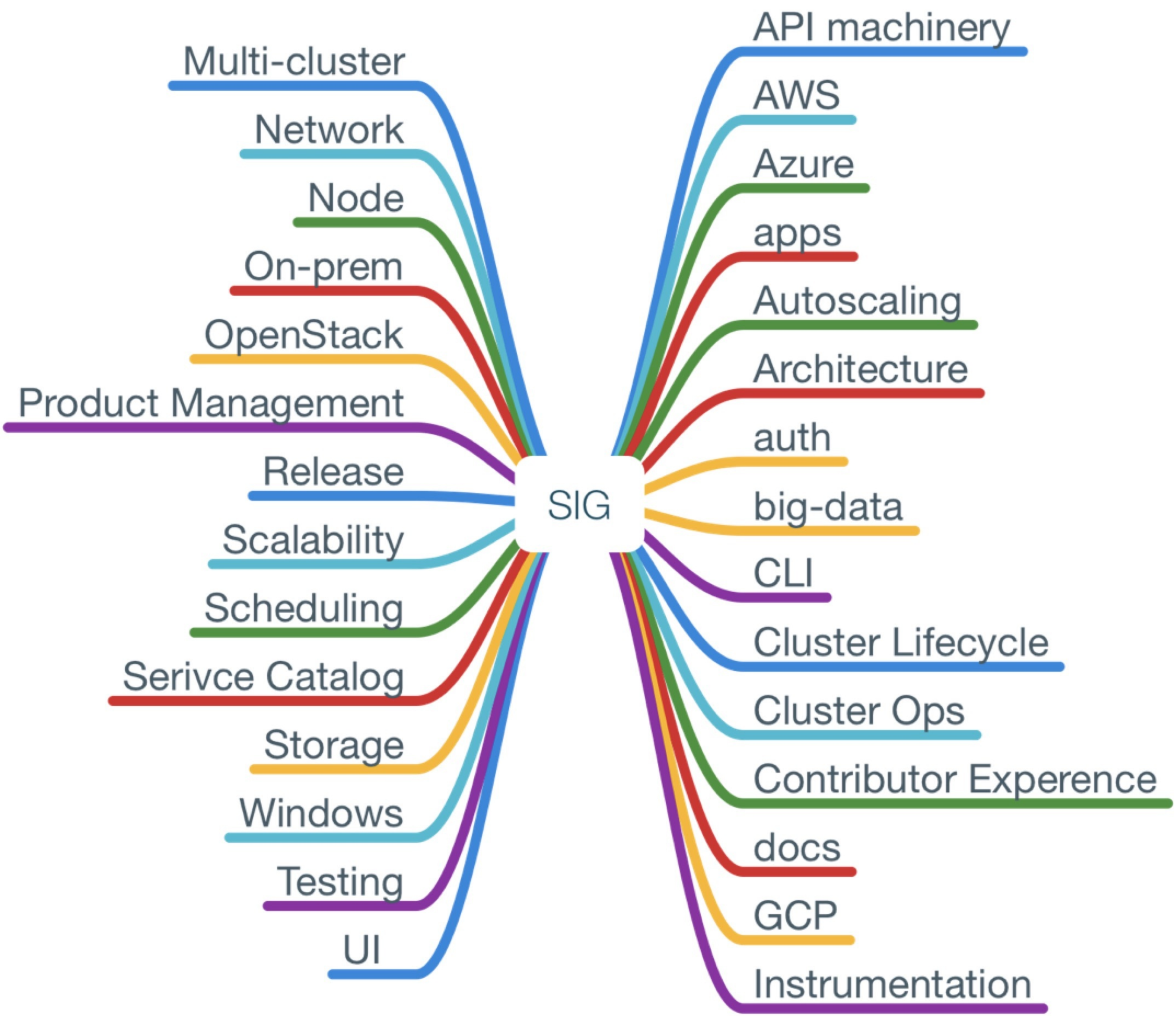
> Contributing is Not only about code, it is about helping a community.

下图是我们刚调研准备使用Kubernetes时候的调研方案选择。



图片 - *Kubernetes solutions*

对于一个初次接触Kubernetes的人来说，看到这样一个庞大的架构选型时会望而生畏，但是Kubernetes的开源社区帮助了我们很多。



图片 - *Kubernetes SIG*

我组建了**K8S&Cloud Native**实战微信群，参与了k8smeetup、KEUC2017、kubernetes-docs-cn Kubernetes官方中文文档项目。

有用的资料和链接

- 我的博客： https://jimmysong.io
- 微信群：k8s&cloud native实战群（见：https://jimmysong.io/about）
- Meetup：k8smeetup
- Cloud Native Go - 基于Go和React云原生Web应用开发：https://jimmysong.io/cloud-native-go
- Gitbook：https://jimmysong.io/kubernetes-handbook
- Cloud native开源生态：https://jimmysong.io/awesome-cloud-native/
- 资料分享整理：https://github.com/rootsongjc/cloud-native-slides-share

- 迁移到云原生架构：https://jimmysong.io/migrating-to-cloud-native-application-architectures/
- KubeCon + CloudNativeCon 2018年11月14-15日 上海

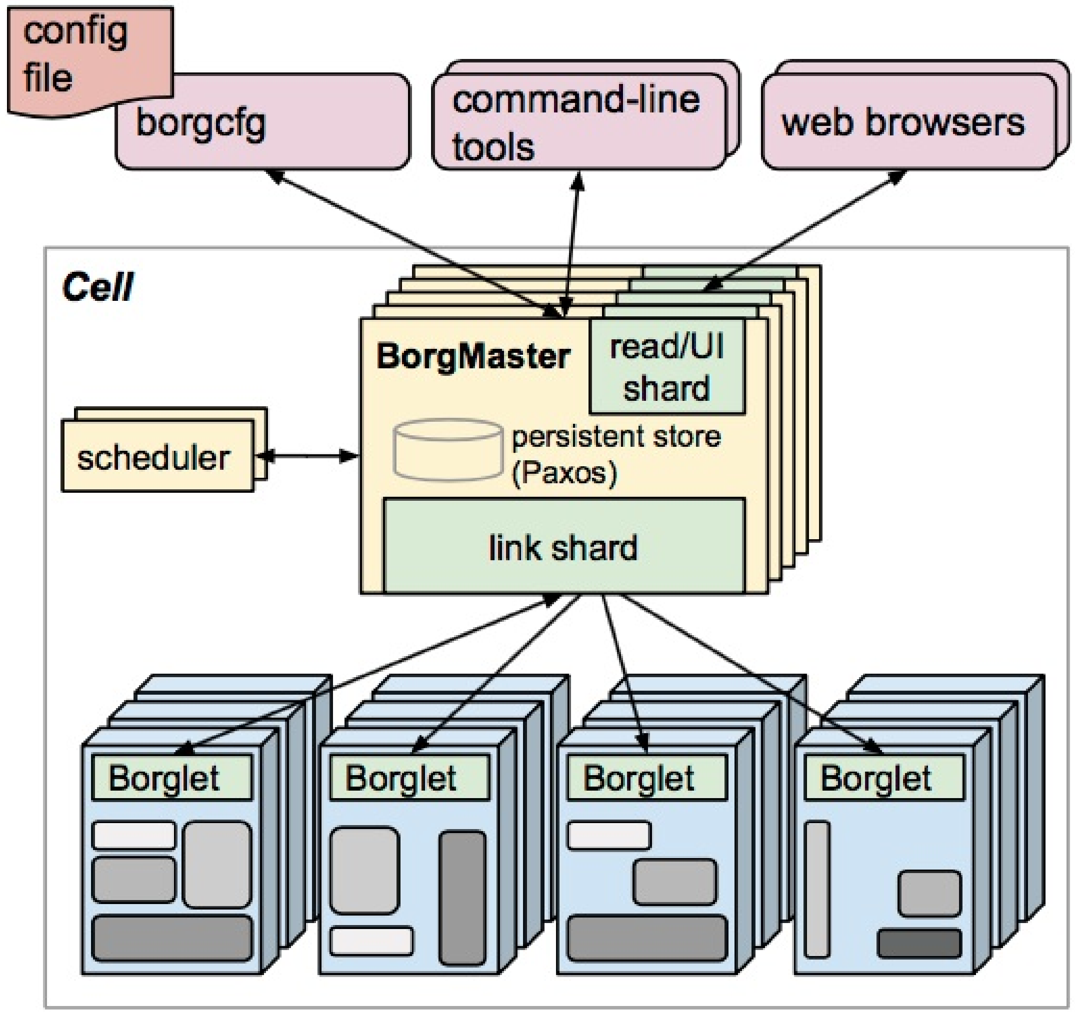
Kubernetes架构

Kubernetes最初源于谷歌内部的Borg，提供了面向应用的容器集群部署和管理系统。Kubernetes的目标旨在消除编排物理/虚拟计算，网络和存储基础设施的负担，并使应用程序运营商和开发人员完全将重点放在以容器为中心的原语上进行自助运营。Kubernetes 也提供稳定、兼容的基础（平台），用于构建定制化的workflows和更高级的自动化任务。 Kubernetes 具备完善的集群管理能力，包括多层次的安全防护和准入机制、多租户应用支撑能力、透明的服务注册和服务发现机制、内建负载均衡器、故障发现和自我修复能力、服务滚动升级和在线扩容、可扩展的资源自动调度机制、多粒度的资源配额管理能力。 Kubernetes 还提供完善的管理工具，涵盖开发、部署测试、运维监控等各个环节。

Borg简介

Borg是谷歌内部的大规模集群管理系统，负责对谷歌内部很多核心服务的调度和管理。Borg的目的是让用户能够不必操心资源管理的问题，让他们专注于自己的核心业务，并且做到跨多个数据中心的资源利用率最大化。

Borg主要由BorgMaster、Borglet、borgcfg和Scheduler组成，如下图所示

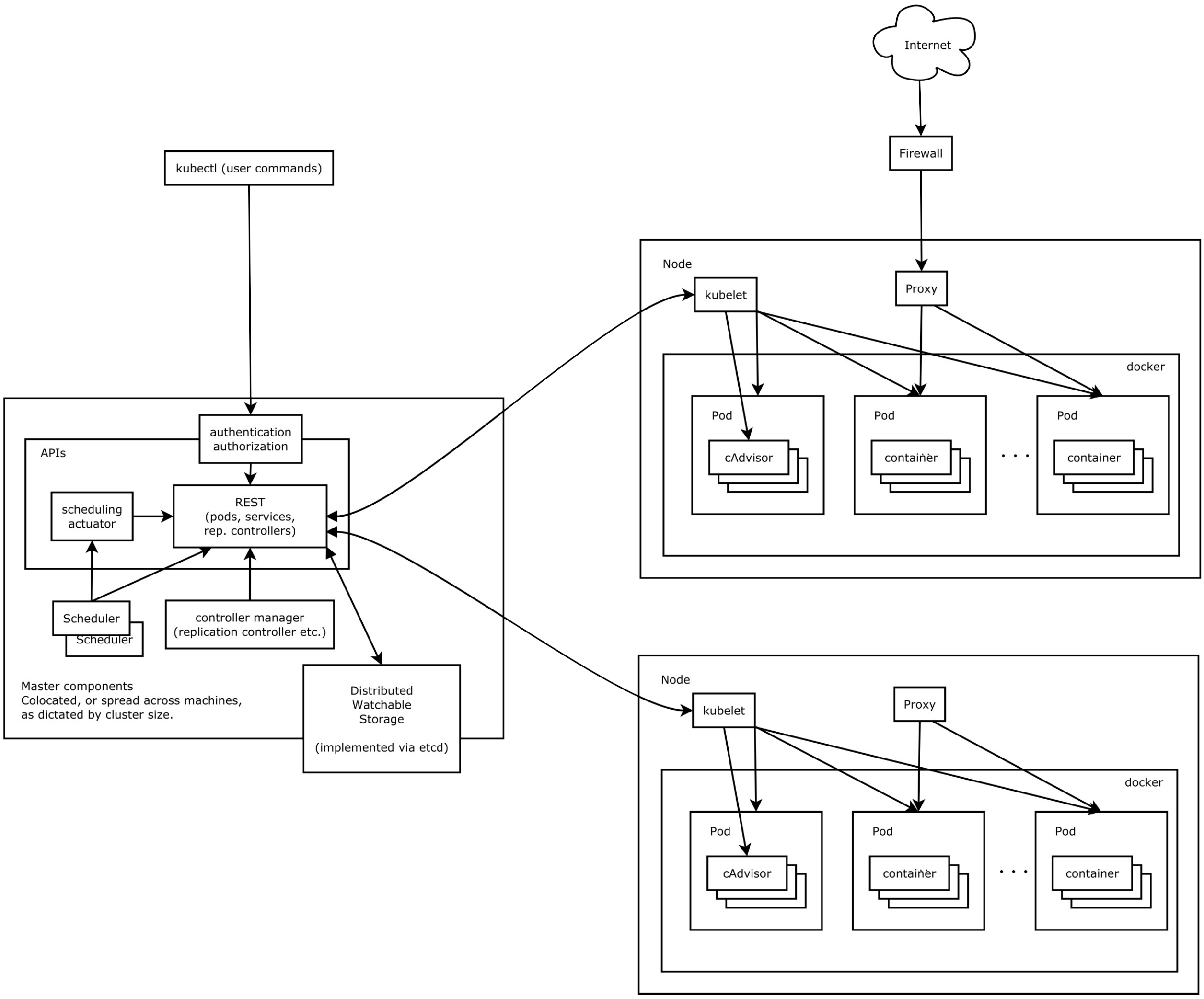


图片 - *Borg*架构

- BorgMaster是整个集群的大脑，负责维护整个集群的状态，并将数据持久化到Paxos存储中；
- Scheduer负责任务的调度，根据应用的特点将其调度到具体的机器上去；
- Borglet负责真正运行任务（在容器中）；
- borgcfg是Borg的命令行工具，用于跟Borg系统交互，一般通过一个配置文件来提交任务。

Kubernetes架构

Kubernetes借鉴了Borg的设计理念，比如Pod、Service、Labels和单Pod单IP等。Kubernetes的整体架构跟Borg非常像，如下图所示



图片 - *Kubernetes*架构

Kubernetes主要由以下几个核心组件组成：

- etcd保存了整个集群的状态；
- apiserver提供了资源操作的唯一入口，并提供认证、授权、访问控制、API注册和发现等机制；
- controller manager负责维护集群的状态，比如故障检测、自动扩展、滚动更新等；
- scheduler负责资源的调度，按照预定的调度策略将Pod调度到相应的机器上；
- kubelet负责维护容器的生命周期，同时也负责Volume（CSI）和网络（CNI）的管理；
- Container runtime负责镜像管理以及Pod和容器的真正运行（CRI）；
- kube-proxy负责为Service提供cluster内部的服务发现和负载均衡；

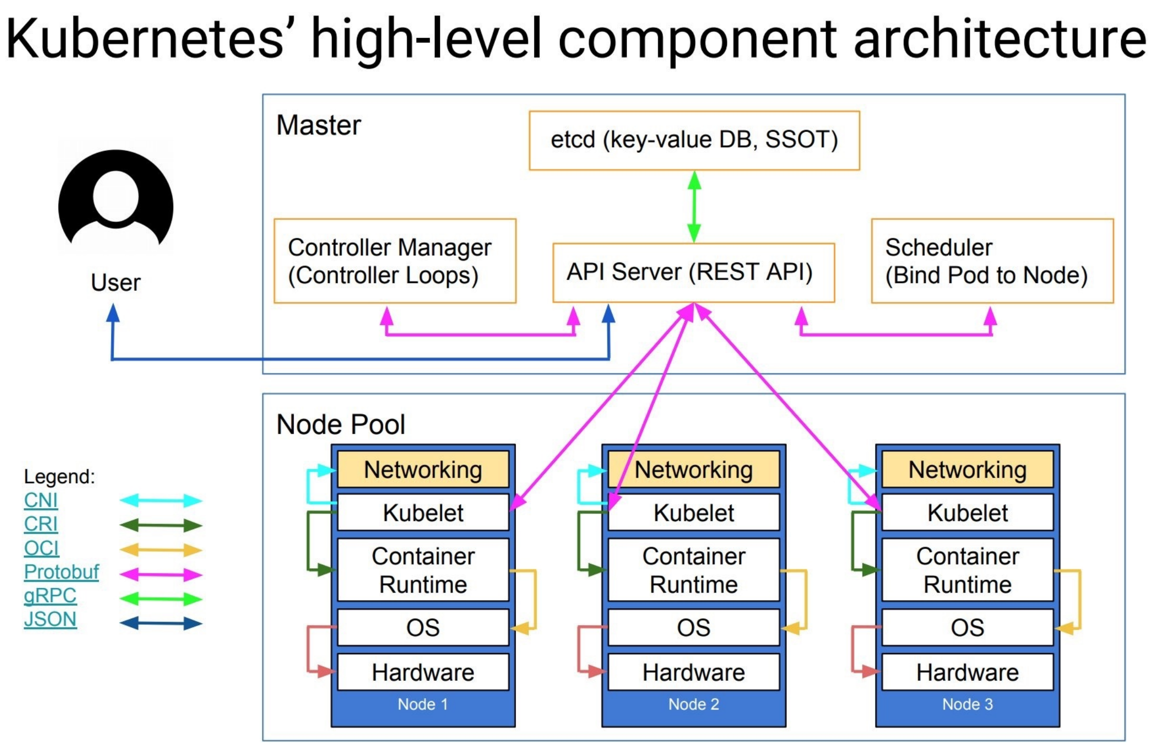
除了核心组件，还有一些推荐的Add-ons：

- kube-dns负责为整个集群提供DNS服务
- Ingress Controller为服务提供外网入口
- Heapster提供资源监控
- Dashboard提供GUI
- Federation提供跨可用区的集群

Kubernetes架构示意图

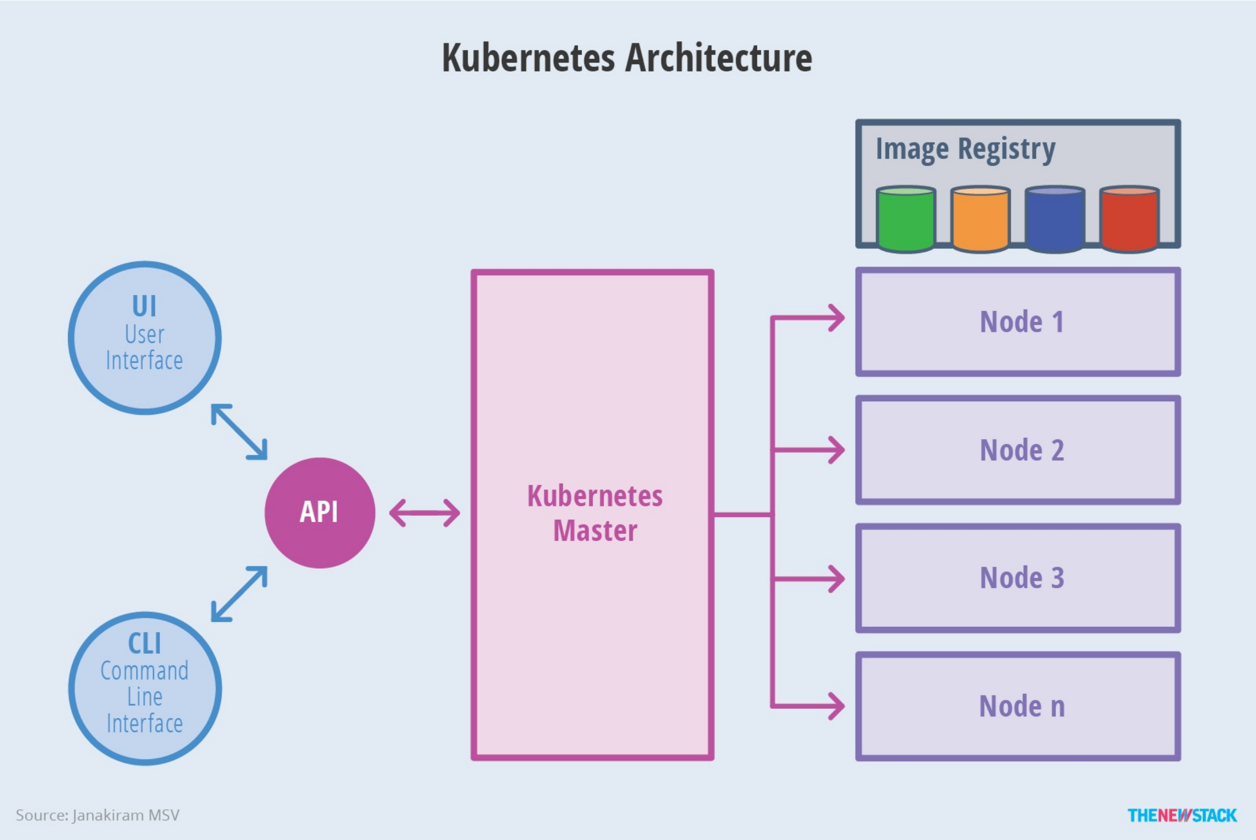
整体架构

下图清晰表明了kubernetes的架构设计以及组件之间的通信协议。



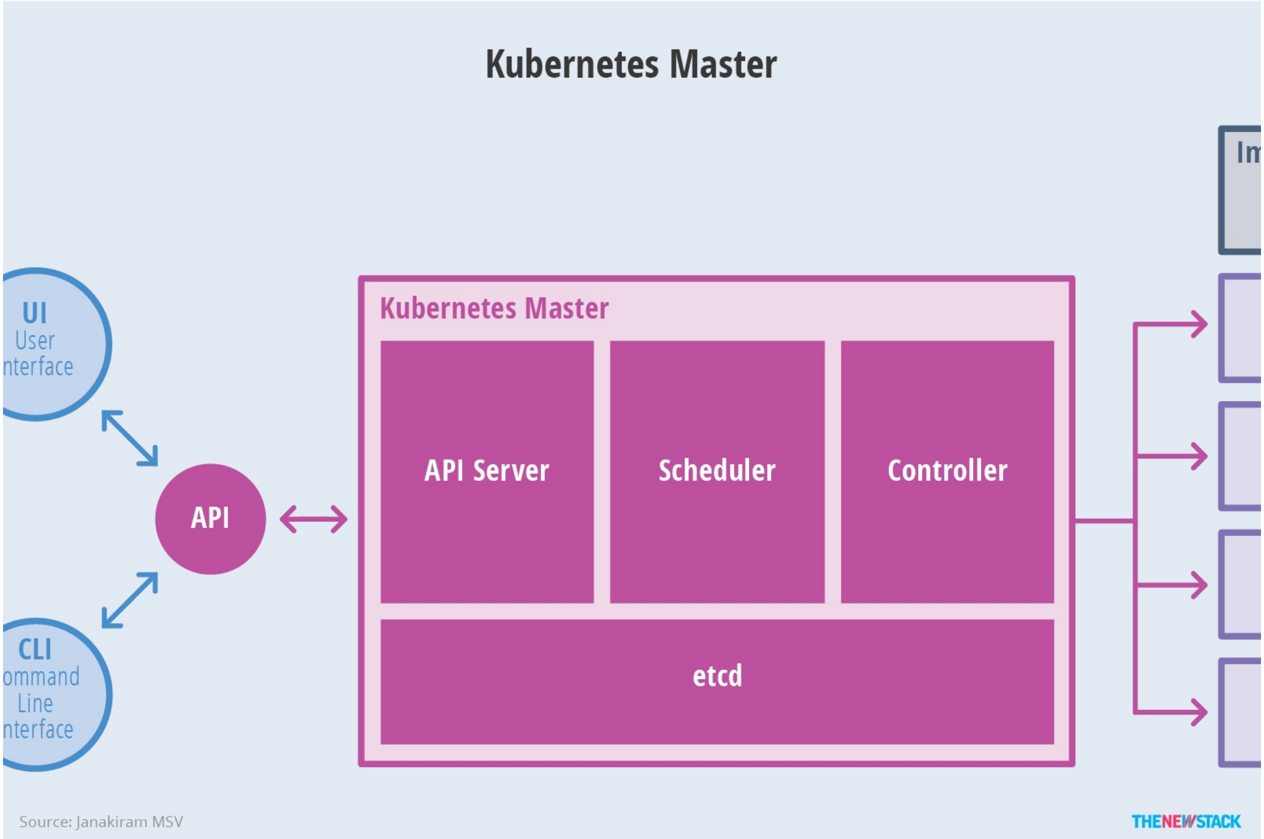
图片 - *Kuberentes*架构（图片来自于网络）

下面是更抽象的一个视图：



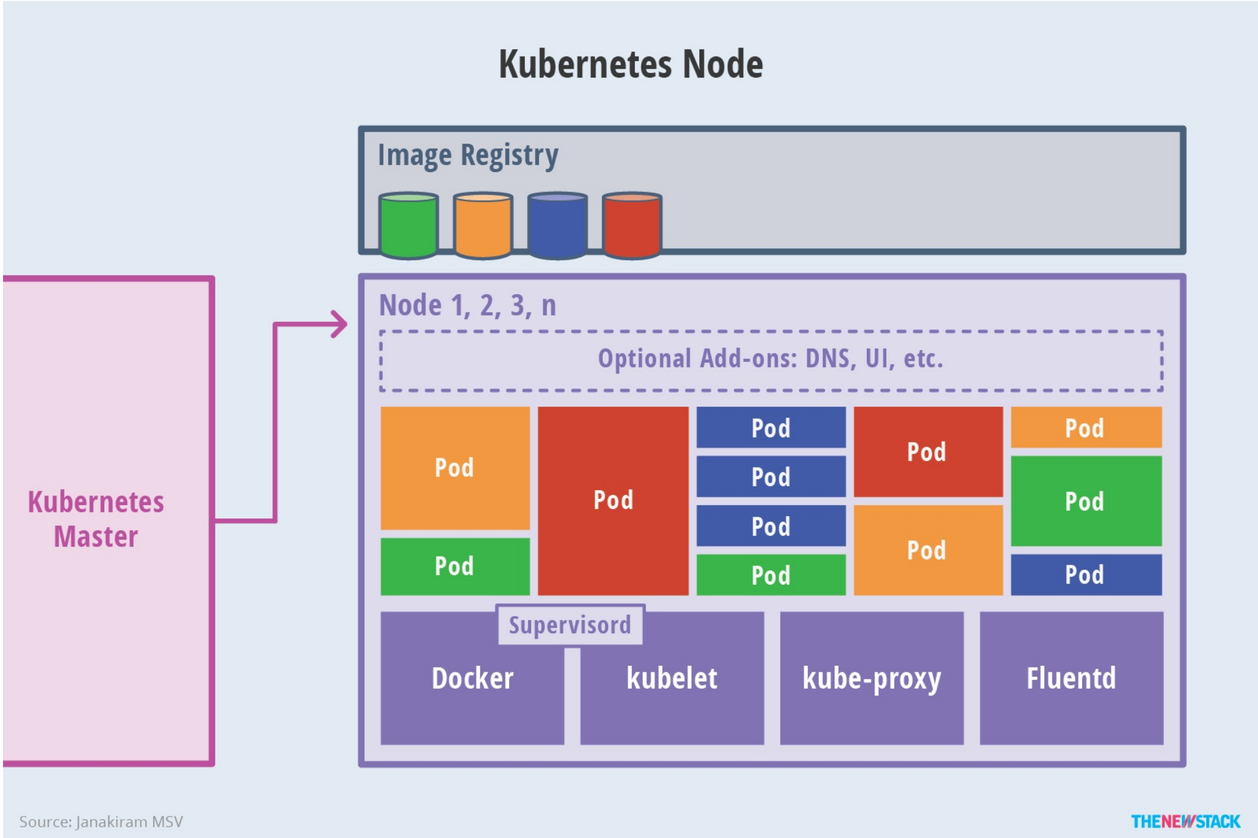
图片 - *kubernetes*整体架构示意图

Master架构



图片 - *Kubernetes master*架构示意图

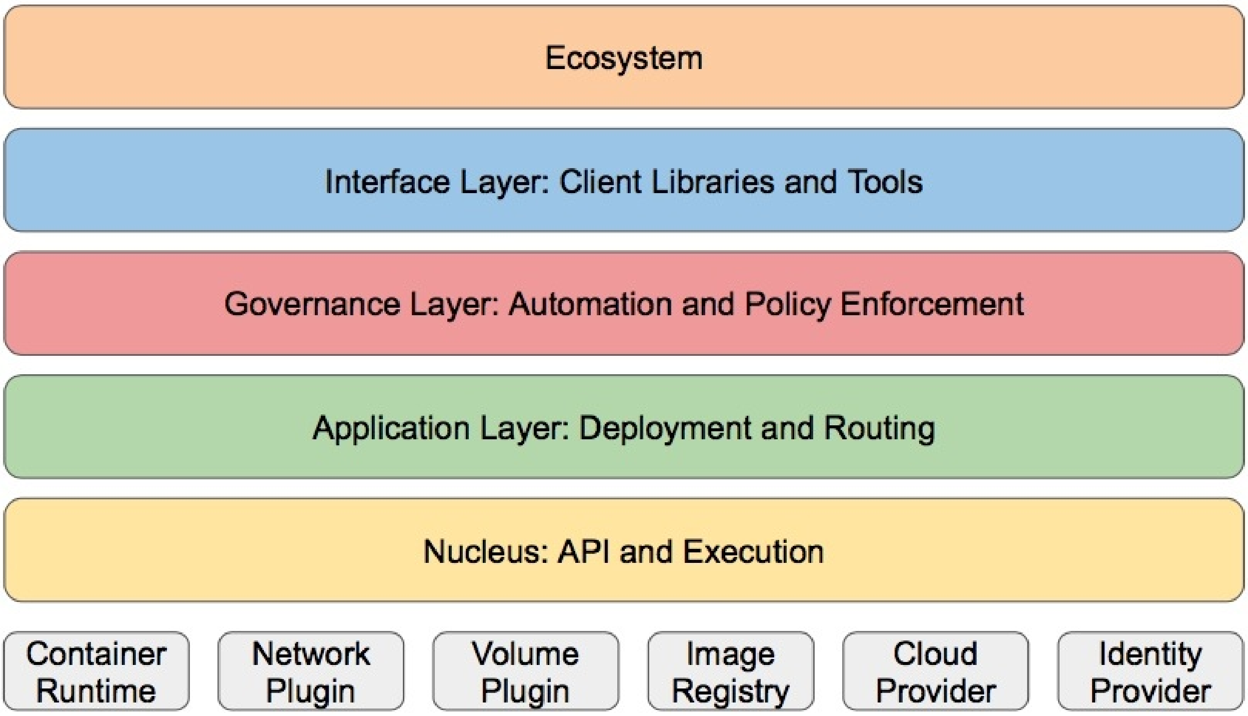
Node架构



图片 - *kubernetes node*架构示意图

分层架构

Kubernetes设计理念和功能其实就是一个类似Linux的分层架构，如下图所示



图片 - *Kubernetes分层架构示意图*

- 核心层：Kubernetes最核心的功能，对外提供API构建高层的应用，对内提供插件式应用执行环境
- 应用层：部署（无状态应用、有状态应用、批处理任务、集群应用等）和路由（服务发现、DNS解析等）
- 管理层：系统度量（如基础设施、容器和网络的度量），自动化（如自动扩展、动态Provision等）以及策略管理（RBAC、Quota、PSP、NetworkPolicy等）
- 接口层：kubectl命令行工具、客户端SDK以及集群联邦
- 生态系统：在接口层之上的庞大容器集群管理调度的生态系统，可以划分为两个范畴
    - Kubernetes外部：日志、监控、配置管理、CI、CD、Workflow、FaaS、OTS应用、ChatOps等
    - Kubernetes内部：CRI、CNI、CVI、镜像仓库、Cloud Provider、集群自身的配置和管理等

> 关于分层架构，可以关注下Kubernetes社区正在推进的Kbernetes architectual roadmap和slide。

参考文档

- Kubernetes design and architecture
- http://queue.acm.org/detail.cfm?id=2898444
- http://static.googleusercontent.com/media/research.google.com/zh-CN//pubs/archive/43438.pdf
- http://thenewstack.io/kubernetes-an-overview
- Kbernetes architectual roadmap和slide

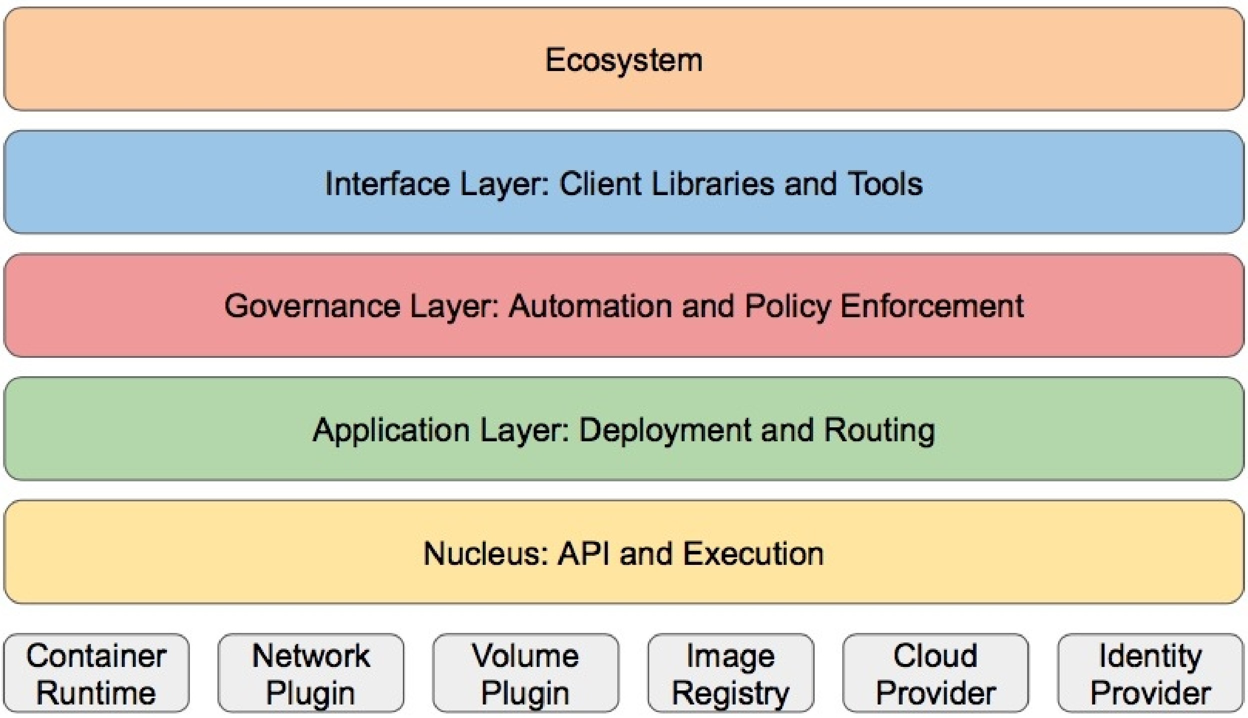
Kubernetes的设计理念

Kubernetes设计理念与分布式系统

分析和理解Kubernetes的设计理念可以使我们更深入地了解Kubernetes系统，更好地利用它管理分布式部署的云原生应用，另一方面也可以让我们借鉴其在分布式系统设计方面的经验。

分层架构

Kubernetes设计理念和功能其实就是一个类似Linux的分层架构，如下图所示



图片 - 分层架构示意图

- 核心层：Kubernetes最核心的功能，对外提供API构建高层的应用，对内提供插件式应用执行环境
- 应用层：部署（无状态应用、有状态应用、批处理任务、集群应用等）和路由（服务发现、DNS解析等）
- 管理层：系统度量（如基础设施、容器和网络的度量），自动化（如自动扩

展、动态Provision等）以及策略管理（RBAC、Quota、PSP、NetworkPolicy等）

- 接口层：kubectl命令行工具、客户端SDK以及集群联邦
- 生态系统：在接口层之上的庞大容器集群管理调度的生态系统，可以划分为两个范畴
  - Kubernetes外部：日志、监控、配置管理、CI、CD、Workflow、FaaS、OTS应用、ChatOps等
  - Kubernetes内部：CRI、CNI、CVI、镜像仓库、Cloud Provider、集群自身的配置和管理等

API设计原则

对于云计算系统，系统API实际上处于系统设计的统领地位，正如本文前面所说，kubernetes集群系统每支持一项新功能，引入一项新技术，一定会新引入对应的API对象，支持对该功能的管理操作，理解掌握的API，就好比抓住了kubernetes系统的牛鼻子。Kubernetes系统API的设计有以下几条原则：

1. **所有API应该是声明式的。**正如前文所说，声明式的操作，相对于命令式操作，对于重复操作的效果是稳定的，这对于容易出现数据丢失或重复的分布式环境来说是很重要的。另外，声明式操作更容易被用户使用，可以使系统向用户隐藏实现的细节，隐藏实现的细节的同时，也就保留了系统未来持续优化的可能性。此外，声明式的API，同时隐含了所有的API对象都是名词性质的，例如Service、Volume这些API都是名词，这些名词描述了用户所期望得到的一个目标分布式对象。
2. **API对象是彼此互补而且可组合的。**这里面实际是鼓励API对象尽量实现面向对象设计时的要求，即“高内聚，松耦合”，对业务相关的概念有一个合适的分解，提高分解出来的对象的可重用性。事实上，Kubernetes这种分布式系统管理平台，也是一种业务系统，只不过它的业务就是调度和管理容器服务。
3. **高层API以操作意图为基础设计。**如何能够设计好API，跟如何能用面向对象的方法设计好应用系统有相通的地方，高层设计一定是从业务出发，而不是过早的从技术实现出发。因此，针对Kubernetes的高层API设计，一定是以Kubernetes的业务为基础出发，也就是以系统调度管理容器的操作意图为基础设计。
4. **低层API根据高层API的控制需要设计。**设计实现低层API的目的，是为了被高层API使用，考虑减少冗余、提高重用性的目的，低层API的设计也要以需求为基础，要尽量抵抗受技术实现影响的诱惑。

5. 尽量避免简单封装，不要有在外部API无法显式知道的内部隐藏的机制。简单的封装，实际没有提供新的功能，反而增加了对所封装API的依赖性。内部隐藏的机制也是非常不利于系统维护的设计方式，例如PetSet和ReplicaSet，本来就是两种Pod集合，那么Kubernetes就用不同API对象来定义它们，而不会说只用同一个ReplicaSet，内部通过特殊的算法再来区分这个ReplicaSet是有状态的还是无状态。
6. API操作复杂度与对象数量成正比。这一条主要是从系统性能角度考虑，要保证整个系统随着系统规模的扩大，性能不会迅速变慢到无法使用，那么最低的限定就是API的操作复杂度不能超过O(N)，N是对象的数量，否则系统就不具备水平伸缩性了。
7. API对象状态不能依赖于网络连接状态。由于众所周知，在分布式环境下，网络连接断开是经常发生的事情，因此要保证API对象状态能应对网络的不稳定，API对象的状态就不能依赖于网络连接状态。
8. 尽量避免让操作机制依赖于全局状态，因为在分布式系统中要保证全局状态的同步是非常困难的。

控制机制设计原则

- 控制逻辑应该只依赖于当前状态。这是为了保证分布式系统的稳定可靠，对于经常出现局部错误的分布式系统，如果控制逻辑只依赖当前状态，那么就非常容易将一个暂时出现故障的系统恢复到正常状态，因为你只要将该系统重置到某个稳定状态，就可以自信的知道系统的所有控制逻辑会开始按照正常方式运行。
- 假设任何错误的可能，并做容错处理。在一个分布式系统中出现局部和临时错误是大概率事件。错误可能来自于物理系统故障，外部系统故障也可能来自于系统自身的代码错误，依靠自己实现的代码不会出错来保证系统稳定其实也是难以实现的，因此要设计对任何可能错误的容错处理。
- 尽量避免复杂状态机，控制逻辑不要依赖无法监控的内部状态。因为分布式系统各个子系统都是不能严格通过程序内部保持同步的，所以如果两个子系统的控制逻辑如果互相有影响，那么子系统就一定要能互相访问到影响控制逻辑的状态，否则，就等同于系统里存在不确定的控制逻辑。
- 假设任何操作都可能被任何操作对象拒绝，甚至被错误解析。由于分布式系统的复杂性以及各子系统的相对独立性，不同子系统经常来自不同的开发团队，所以不能奢望任何操作被另一个子系统以正确的方式处理，要保证出现错误的时候，操作级别的错误不会影响到系统稳定性。
- 每个模块都可以在出错后自动恢复。由于分布式系统中无法保证系统各个模块

是始终连接的，因此每个模块要有自我修复的能力，保证不会因为连接不到其他模块而自我崩溃。

- 每个模块都可以在必要时优雅地降级服务。所谓优雅地降级服务，是对系统鲁棒性的要求，即要求在设计实现模块时划分清楚基本功能和高级功能，保证基本功能不会依赖高级功能，这样同时就保证了不会因为高级功能出现故障而导致整个模块崩溃。根据这种理念实现的系统，也更容易快速地增加新的高级功能，因为不必担心引入高级功能影响原有的基本功能。

Kubernetes的核心技术概念和API对象

API对象是Kubernetes集群中的管理操作单元。Kubernetes集群系统每支持一项新功能，引入一项新技术，一定会新引入对应的API对象，支持对该功能的管理操作。例如副本集Replica Set对应的API对象是RS。

每个API对象都有3大类属性：元数据metadata、规范spec和状态status。元数据是用来标识API对象的，每个对象都至少有3个元数据：namespace，name和uid；除此以外还有各种各样的标签labels用来标识和匹配不同的对象，例如用户可以用标签env来标识区分不同的服务部署环境，分别用env=dev、env=testing、env=production来标识开发、测试、生产的不同服务。规范描述了用户期望Kubernetes集群中的分布式系统达到的理想状态（Desired State），例如用户可以通过复制控制器Replication Controller设置期望的Pod副本数为3；status描述了系统实际当前达到的状态（Status），例如系统当前实际的Pod副本数为2；那么复制控制器当前的程序逻辑就是自动启动新的Pod，争取达到副本数为3。

Kubernetes中所有的配置都是通过API对象的spec去设置的，也就是用户通过配置系统的理想状态来改变系统，这是Kubernetes重要设计理念之一，即所有的操作都是声明式（Declarative）的而不是命令式（Imperative）的。声明式操作在分布式系统中的好处是稳定，不怕丢操作或运行多次，例如设置副本数为3的操作运行多次也还是一个结果，而给副本数加1的操作就不是声明式的，运行多次结果就错了。

Pod

Kubernetes有很多技术概念，同时对应很多API对象，最重要的也是最基础的是Pod。Pod是在Kubernetes集群中运行部署应用或服务的最小单元，它是可以支持多容器的。Pod的设计理念是支持多个容器在一个Pod中共享网络地址和文件系

统，可以通过进程间通信和文件共享这种简单高效的方式组合完成服务。Pod对多容器的支持是K8最基础的设计理念。比如你运行一个操作系统发行版的软件仓库，一个Nginx容器用来发布软件，另一个容器专门用来从源仓库做同步，这两个容器的镜像不太可能是一个团队开发的，但是他们一块儿工作才能提供一个微服务；这种情况下，不同的团队各自开发构建自己的容器镜像，在部署的时候组合成一个微服务对外提供服务。

Pod是Kubernetes集群中所有业务类型的基础，可以看作运行在K8集群中的小机器人，不同类型的业务就需要不同类型的小机器人去执行。目前Kubernetes中的业务主要可以分为长期伺服型（long-running）、批处理型（batch）、节点后台支撑型（node-daemon）和有状态应用型（stateful application）；分别对应的小机器人控制器为Deployment、Job、DaemonSet和PetSet，本文后面会一一介绍。

副本控制器（Replication Controller，RC）

RC是Kubernetes集群中最早的保证Pod高可用的API对象。通过监控运行中的Pod来保证集群中运行指定数目的Pod副本。指定的数目可以是多个也可以是1个；少于指定数目，RC就会启动运行新的Pod副本；多于指定数目，RC就会杀死多余的Pod副本。即使在指定数目为1的情况下，通过RC运行Pod也比直接运行Pod更明智，因为RC也可以发挥它高可用的能力，保证永远有1个Pod在运行。RC是Kubernetes较早期的技术概念，只适用于长期伺服型的业务类型，比如控制小机器人提供高可用的Web服务。

副本集（Replica Set，RS）

RS是新一代RC，提供同样的高可用能力，区别主要在于RS后来居上，能支持更多种类的匹配模式。副本集对象一般不单独使用，而是作为Deployment的理想状态参数使用。

部署（Deployment）

部署表示用户对Kubernetes集群的一次更新操作。部署是一个比RS应用模式更广的API对象，可以是创建一个新的服务，更新一个新的服务，也可以是滚动升级一个服务。滚动升级一个服务，实际是创建一个新的RS，然后逐渐将新RS中副本数

增加到理想状态，将旧RS中的副本数减小到0的复合操作；这样一个复合操作用一个RS是不太好描述的，所以用一个更通用的Deployment来描述。以Kubernetes的发展方向，未来对所有长期伺服型的的业务的管理，都会通过Deployment来管理。

服务（Service）

RC、RS和Deployment只是保证了支撑服务的微服务Pod的数量，但是没有解决如何访问这些服务的问题。一个Pod只是一个运行服务的实例，随时可能在一个节点上停止，在另一个节点以一个新的IP启动一个新的Pod，因此不能以确定的IP和端口号提供服务。要稳定地提供服务需要服务发现和负载均衡能力。服务发现完成的工作，是针对客户端访问的服务，找到对应的的后端服务实例。在K8集群中，客户端需要访问的服务就是Service对象。每个Service会对应一个集群内部有效的虚拟IP，集群内部通过虚拟IP访问一个服务。在Kubernetes集群中微服务的负载均衡是由Kube-proxy实现的。Kube-proxy是Kubernetes集群内部的负载均衡器。它是一个分布式代理服务器，在Kubernetes的每个节点上都有一个；这一设计体现了它的伸缩性优势，需要访问服务的节点越多，提供负载均衡能力的Kube-proxy就越多，高可用节点也随之增多。与之相比，我们平时在服务器端做个反向代理做负载均衡，还要进一步解决反向代理的负载均衡和高可用问题。

任务（Job）

Job是Kubernetes用来控制批处理型任务的API对象。批处理业务与长期伺服业务的主要区别是批处理业务的运行有头有尾，而长期伺服业务在用户不停止的情况下永远运行。Job管理的Pod根据用户的设置把任务成功完成就自动退出了。成功完成的标志根据不同的spec.completions策略而不同：单Pod型任务有一个Pod成功就标志完成；定数成功型任务保证有N个任务全部成功；工作队列型任务根据应用确认的全局成功而标志成功。

后台支撑服务集（DaemonSet）

长期伺服型和批处理型服务的核心在业务应用，可能有些节点运行多个同类业务的Pod，有些节点上又没有这类Pod运行；而后台支撑型服务的核心关注点在Kubernetes集群中的节点（物理机或虚拟机），要保证每个节点上都有一个此类Pod运行。节点可能是所有集群节点也可能是通过nodeSelector选定的一些特定节点。典型的后台支撑型服务包括，存储，日志和监控等在每个节点上支持Kubernetes集群运行的服务。

有状态服务集（PetSet）

Kubernetes在1.3版本里发布了Alpha版的PetSet功能。在云原生应用的体系里，有下面两组近义词；第一组是无状态（stateless）、牲畜（cattle）、无名（nameless）、可丢弃（disposable）；第二组是有状态（stateful）、宠物（pet）、有名（having name）、不可丢弃（non-disposable）。RC和RS主要是控制提供无状态服务的，其所控制的Pod的名字是随机设置的，一个Pod出故障了就被丢弃掉，在另一个地方重启一个新的Pod，名字变了、名字和启动在哪儿都不重要，重要的只是Pod总数；而PetSet是用来控制有状态服务，PetSet中的每个Pod的名字都是事先确定的，不能更改。PetSet中Pod的名字的作用，并不是《千与千寻》的人性原因，而是关联与该Pod对应的状态。

对于RC和RS中的Pod，一般不挂载存储或者挂载共享存储，保存的是所有Pod共享的状态，Pod像牲畜一样没有分别（这似乎也确实意味着失去了人性特征）；对于PetSet中的Pod，每个Pod挂载自己独立的存储，如果一个Pod出现故障，从其他节点启动一个同样名字的Pod，要挂载上原来Pod的存储继续以它的状态提供服务。

适合于PetSet的业务包括数据库服务MySQL和PostgreSQL，集群化管理服务Zookeeper、etcd等有状态服务。PetSet的另一种典型应用场景是作为一种比普通容器更稳定可靠的模拟虚拟机的机制。传统的虚拟机正是一种有状态的宠物，运维人员需要不断地维护它，容器刚开始流行时，我们用容器来模拟虚拟机使用，所有状态都保存在容器里，而这已被证明是非常不安全、不可靠的。使用PetSet，Pod仍然可以通过漂移到不同节点提供高可用，而存储也可以通过外挂的存储来提供高可靠性，PetSet做的只是将确定的Pod与确定的存储关联起来保证状态的连续性。PetSet还只在Alpha阶段，后面的设计如何演变，我们还要继续观察。

集群联邦（Federation）

Kubernetes在1.3版本里发布了beta版的Federation功能。在云计算环境中，服务的作用距离范围从近到远一般可以有：同主机（Host，Node）、跨主机同可用区（Available Zone）、跨可用区同地区（Region）、跨地区同服务商（Cloud Service Provider）、跨云平台。Kubernetes的设计定位是单一集群在同一个地域内，因为同一个地区的网络性能才能满足Kubernetes的调度和计算存储连接要求。而联合集群服务就是为提供跨Region跨服务商Kubernetes集群服务而设计的。

每个Kubernetes Federation有自己的分布式存储、API Server和Controller Manager。用户可以通过Federation的API Server注册该Federation的成员Kubernetes Cluster。当用户通过Federation的API Server创建、更改API对象时，Federation API Server会在自己所有注册的子Kubernetes Cluster都创建一份对应的API对象。在提供业务请求服务时，Kubernetes Federation会先在自己的各个子Cluster之间做负载均衡，而对于发送到某个具体Kubernetes Cluster的业务请求，会依照这个Kubernetes Cluster独立提供服务时一样的调度模式去做Kubernetes Cluster内部的负载均衡。而Cluster之间的负载均衡是通过域名服务的负载均衡来实现的。

所有的设计都尽量不影响Kubernetes Cluster现有的工作机制，这样对于每个子Kubernetes集群来说，并不需要更外层的有一个Kubernetes Federation，也就是意味着所有现有的Kubernetes代码和机制不需要因为Federation功能有任何变化。

存储卷（Volume）

Kubernetes集群中的存储卷跟Docker的存储卷有些类似，只不过Docker的存储卷作用范围为一个容器，而Kubernetes的存储卷的生命周期和作用范围是一个Pod。每个Pod中声明的存储卷由Pod中的所有容器共享。Kubernetes支持非常多的存储卷类型，特别的，支持多种公有云平台的存储，包括AWS，Google和Azure云；支持多种分布式存储包括GlusterFS和Ceph；也支持较容易使用的主机本地目录hostPath和NFS。Kubernetes还支持使用Persistent Volume Claim即PVC这种逻辑存储，使用这种存储，使得存储的使用者可以忽略后台的实际存储技术（例如AWS，Google或GlusterFS和Ceph），而将有关存储实际技术的配置交给存储管理员通过Persistent Volume来配置。

持久存储卷（Persistent Volume，PV）和持久存储卷声明（Persistent Volume Claim，PVC）

PV和PVC使得Kubernetes集群具备了存储的逻辑抽象能力，使得在配置Pod的逻辑里可以忽略对实际后台存储技术的配置，而把这项配置的工作交给PV的配置者，即集群的管理者。存储的PV和PVC的这种关系，跟计算的Node和Pod的关系是非常类似的；PV和Node是资源的提供者，根据集群的基础设施变化而变化，由Kubernetes集群管理员配置；而PVC和Pod是资源的使用者，根据业务服务的需求变化而变化，有Kubernetes集群的使用者即服务的管理员来配置。

节点（Node）

Kubernetes集群中的计算能力由Node提供，最初Node称为服务节点Minion，后来改名为Node。Kubernetes集群中的Node也就等同于Mesos集群中的Slave节点，是所有Pod运行所在的工作主机，可以是物理机也可以是虚拟机。不论是物理机还是虚拟机，工作主机的统一特征是上面要运行kubelet管理节点上运行的容器。

密钥对象（Secret）

Secret是用来保存和传递密码、密钥、认证凭证这些敏感信息的对象。使用Secret的好处是可以避免把敏感信息明文写在配置文件里。在Kubernetes集群中配置和使用服务不可避免的要用到各种敏感信息实现登录、认证等功能，例如访问AWS存储的用户名密码。为了避免将类似的敏感信息明文写在所有需要使用的配置文件中，可以将这些信息存入一个Secret对象，而在配置文件中通过Secret对象引用这些敏感信息。这种方式的好处包括：意图明确，避免重复，减少暴漏机会。

用户帐户（User Account）和服务帐户（Service Account）

顾名思义，用户帐户为人提供账户标识，而服务账户为计算机进程和Kubernetes集群中运行的Pod提供账户标识。用户帐户和服务帐户的一个区别是作用范围；用户帐户对应的是人的身份，人的身份与服务的namespace无关，所以用户账户是跨namespace的；而服务帐户对应的是一个运行中程序的身份，与特定namespace是相关的。

命名空间（Namespace）

命名空间为Kubernetes集群提供虚拟的隔离作用，Kubernetes集群初始有两个命名空间，分别是默认命名空间default和系统命名空间kube-system，除此以外，管理员可以可以创建新的命名空间满足需要。

RBAC访问授权

Kubernetes在1.3版本中发布了alpha版的基于角色的访问控制（Role-based Access Control，RBAC）的授权模式。相对于基于属性的访问控制（Attribute-based Access Control，ABAC），RBAC主要是引入了角色（Role）和角色绑定

（RoleBinding）的抽象概念。在ABAC中，Kubernetes集群中的访问策略只能跟用户直接关联；而在RBAC中，访问策略可以跟某个角色关联，具体的用户在跟一个或多个角色相关联。显然，RBAC像其他新功能一样，每次引入新功能，都会引入新的API对象，从而引入新的概念抽象，而这一新的概念抽象一定会使集群服务管理和使用更容易扩展和重用。

总结

从Kubernetes的系统架构、技术概念和设计理念，我们可以看到Kubernetes系统最核心的两个设计理念：一个是容错性，一个是易扩展性。容错性实际是保证Kubernetes系统稳定性和安全性的基础，易扩展性是保证Kubernetes对变更友好，可以快速迭代增加新功能的基础。

按照分布式系统一致性算法Paxos发明人计算机科学家Leslie Lamport的理念，一个分布式系统有两类特性：安全性Safety和活性Liveness。安全性保证系统的稳定，保证系统不会崩溃，不会出现业务错误，不会做坏事，是严格约束的；活性使得系统可以提供功能，提高性能，增加易用性，让系统可以在用户“看到的时间内”做些好事，是尽力而为的。Kubernetes系统的设计理念正好与Lamport安全性与活性的理念不谋而合，也正是因为Kubernetes在引入功能和技术的时候，非常好地划分了安全性和活性，才可以让Kubernetes能有这么快版本迭代，快速引入像RBAC、Federation和PetSet这种新功能。

原文地址：《Kubernetes与云原生应用》系列之Kubernetes的系统架构与设计理念

Etcd解析

Etcd是Kubernetes集群中的一个十分重要的组件，用于保存集群所有的网络配置和对象的状态信息。在后面具体的安装环境中，我们安装的etcd的版本是v3.1.5，整个kubernetes系统中一共有两个服务需要用到etcd用来协同和存储配置，分别是：

- 网络插件flannel、对于其它网络插件也需要用到etcd存储网络的配置信息
- kubernetes本身，包括各种对象的状态和元信息配置

注意：flannel操作etcd使用的是v2的API，而kubernetes操作etcd使用的v3的API，所以在下面我们执行 `etcdctl` 的时候需要设置 `ETCDCTL_API` 环境变量，该变量默认值为2。

原理

Etcd使用的是raft一致性算法来实现的，是一款分布式的一致性KV存储，主要用于共享配置和服务发现。关于raft一致性算法请参考该动画演示。

关于Etcd的原理解析请参考Etcd 架构与实现解析。

使用Etcd存储Flannel网络信息

我们在安装Flannel的时候配置了 `FLANNEL_ETCD_PREFIX="/kube-centos/network"` 参数，这是Flannel查询etcd的目录地址。

查看Etcd中存储的flannel网络信息：

```
$ etcdctl --ca-file=/etc/kubernetes/ssl/ca.pem --cert-file=/etc/kubernetes/ssl/kubernetes.pem --key-file=/etc/kubernetes/ssl/kubernetes-key.pem ls /kube-centos/network -r
2018-01-19 18:38:22.768145 I | warning: ignoring ServerName for user-provided CA for backwards compatibility is deprecated
/kube-centos/network/config
/kube-centos/network/subnets
/kube-centos/network/subnets/172.30.31.0-24
/kube-centos/network/subnets/172.30.20.0-24
/kube-centos/network/subnets/172.30.23.0-24
```

查看flannel的配置：

```
$ etcdctl --ca-file=/etc/kubernetes/ssl/ca.pem --cert-file=/etc/
kubernetes/ssl/kubernetes.pem --key-file=/etc/kubernetes/ssl/kub
ernetes-key.pem get /kube-centos/network/config
2018-01-19 18:38:22.768145 I | warning: ignoring ServerName for
user-provided CA for backwards compatibility is deprecated
{ "Network": "172.30.0.0/16", "SubnetLen": 24, "Backend": { "Typ
e": "host-gw" } }
```

使用Etcd存储Kubernetes对象信息

Kubernetes使用etcd v3的API操作etcd中的数据。所有的资源对象都保存在 `/registry` 路径下，如下：

```
ThirdPartyResourceData
apiextensions.k8s.io
apiregistration.k8s.io
certificatesigningrequests
clusterrolebindings
clusterroles
configmaps
controllerrevisions
controllers
daemonsets
deployments
events
horizontalpodautoscalers
ingress
limitranges
minions
monitoring.coreos.com
namespaces
persistentvolumeclaims
persistentvolumes
poddisruptionbudgets
pods
ranges
replicasets
resourcequotas
rolebindings
roles
secrets
serviceaccounts
services
statefulsets
storageclasses
thirdpartyresources
```

如果你还创建了CRD（自定义资源定义），则在此会出现CRD的API。

查看集群中所有的Pod信息

例如我们直接从etcd中查看kubernetes集群中所有的pod的信息，可以使用下面的命令：

```
ETCDCTL_API=3 etcdctl get /registry/pods --prefix -w json|python
 -m json.tool
```

此时将看到json格式输出的结果，其中的 `key` 使用了 `base64` 编码，关于etcdctl命令的详细用法请参考使用etcdctl访问kubernetes数据。

Etcd V2与V3版本API的区别

Etcd V2和V3之间的数据结构完全不同，互不兼容，也就是说使用V2版本的API创建的数据只能使用V2的API访问，V3的版本的API创建的数据只能使用V3的API访问。这就造成我们访问etcd中保存的flannel的数据需要使用 `etcdctl` 的V2版本的客户端，而访问kubernetes的数据需要设置 `ETCDCTL_API=3` 环境变量来指定V3版本的API。

Etcd数据备份

我们安装的时候指定的Etcd数据的存储路径是 `/var/lib/etcd` ，一定要对该目录做好备份。

参考

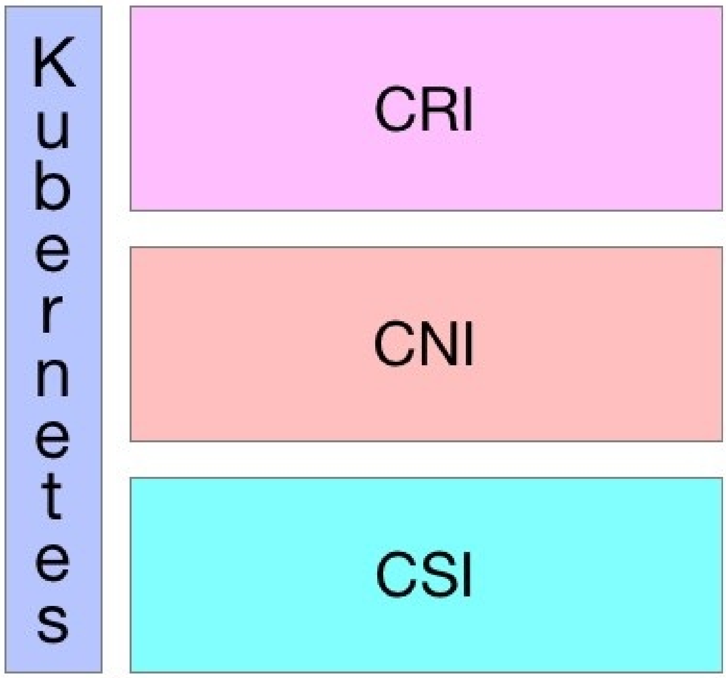
- etcd官方文档
- etcd v3命令和API
- Etcd 架构与实现解析

开放接口

Kubernetes作为云原生应用的的基础调度平台，相当于云原生的操作系统，为了便于系统的扩展，Kubernetes中开放的以下接口，可以分别对接不同的后端，来实现自己的业务逻辑：

- **CRI**（**Container Runtime Interface**）：容器运行时接口，提供计算资源
- **CNI**（**Container Network Interface**）：容器网络接口，提供网络资源
- **CSI**（**Container Storage Interface**）：容器存储接口，提供存储资源

以上三种资源相当于一个分布式操作系统的最基础的几种资源类型，而Kuberentes是将他们粘合在一起的纽带。

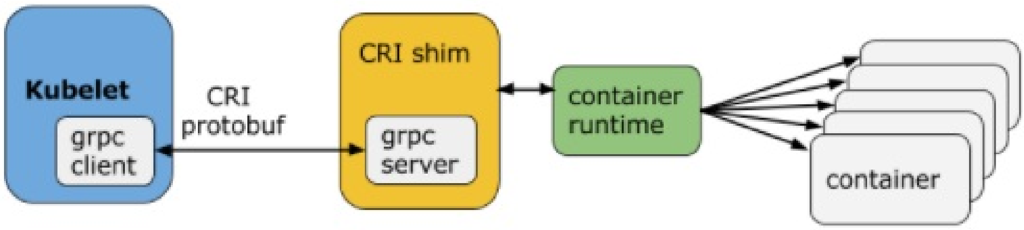


CRI - Container Runtime Interface（容器运行时接口）

CRI中定义了容器和镜像的服务的接口，因为容器运行时与镜像的生命周期是彼此隔离的，因此需要定义两个服务。该接口使用Protocol Buffer，基于gRPC，在kubernetes v1.7+版本中是在pkg/kubelet/apis/cri/v1alpha1/runtime的 `api.proto` 中定义的。

CRI架构

Container Runtime实现了CRI gRPC Server，包括 `RuntimeService` 和 `ImageService` 。该gRPC Server需要监听本地的Unix socket，而kubelet则作为gRPC Client运行。



图片 - *CRI*架构-图片来自*kubernetes blog*

启用CRI

除非集成了rktnetes，否则CRI都是被默认启用了，kubernetes1.7版本开始旧的预集成的docker CRI已经被移除。

要想启用CRI只需要在kubelet的启动参数重传入此参数： `--container-runtime-endpoint` 远程运行时服务的端点。当前Linux上支持unix socket，windows上支持tcp。例如： `unix:///var/run/dockershim.sock` 、 `tcp://localhost:373` ，默认是 `unix:///var/run/dockershim.sock` ，即默认使用本地的docker作为容器运行时。

关于CRI的详细进展请参考CRI: the Container Runtime Interface。

CRI接口

Kubernetes1.9中的CRI接口在 `api.proto` 中的定义如下：

```
// Runtime service defines the public APIs for remote container
runtimes
service RuntimeService {
    // Version returns the runtime name, runtime version, and ru
ntime API version.
    rpc Version(VersionRequest) returns (VersionResponse) {}

    // RunPodSandbox creates and starts a pod-level sandbox. Run
times must ensure
    // the sandbox is in the ready state on success.
    rpc RunPodSandbox(RunPodSandboxRequest) returns (RunPodSandb
oxResponse) {}
    // StopPodSandbox stops any running process that is part of
the sandbox and
    // reclaims network resources (e.g., IP addresses) allocated
 to the sandbox.
    // If there are any running containers in the sandbox, they
must be forcibly
    // terminated.
    // This call is idempotent, and must not return an error if
all relevant
    // resources have already been reclaimed. kubelet will call
StopPodSandbox
    // at least once before calling RemovePodSandbox. It will al
so attempt to
    // reclaim resources eagerly, as soon as a sandbox is not ne
eded. Hence,
    // multiple StopPodSandbox calls are expected.
    rpc StopPodSandbox(StopPodSandboxRequest) returns (StopPodSa
ndboxResponse) {}
    // RemovePodSandbox removes the sandbox. If there are any ru
nning containers
    // in the sandbox, they must be forcibly terminated and remo
ved.
    // This call is idempotent, and must not return an error if
the sandbox has
    // already been removed.
    rpc RemovePodSandbox(RemovePodSandboxRequest) returns (Remov
ePodSandboxResponse) {}
    // PodSandboxStatus returns the status of the PodSandbox. If
 the PodSandbox is not
    // present, returns an error.
    rpc PodSandboxStatus(PodSandboxStatusRequest) returns (PodSa
ndboxStatusResponse) {}
    // ListPodSandbox returns a list of PodSandboxes.
    rpc ListPodSandbox(ListPodSandboxRequest) returns (ListPodSa
ndboxResponse) {}
```

```
    // CreateContainer creates a new container in specified PodS
andbox
    rpc CreateContainer(CreateContainerRequest) returns (CreateC
ontainerResponse) {}
    // StartContainer starts the container.
    rpc StartContainer(StartContainerRequest) returns (StartCont
ainerResponse) {}
    // StopContainer stops a running container with a grace peri
od (i.e., timeout).
    // This call is idempotent, and must not return an error if
the container has
    // already been stopped.
    // TODO: what must the runtime do after the grace period is
reached?
    rpc StopContainer(StopContainerRequest) returns (StopContain
erResponse) {}
    // RemoveContainer removes the container. If the container i
s running, the
    // container must be forcibly removed.
    // This call is idempotent, and must not return an error if
the container has
    // already been removed.
    rpc RemoveContainer(RemoveContainerRequest) returns (RemoveC
ontainerResponse) {}
    // ListContainers lists all containers by filters.
    rpc ListContainers(ListContainersRequest) returns (ListConta
inersResponse) {}
    // ContainerStatus returns status of the container. If the c
ontainer is not
    // present, returns an error.
    rpc ContainerStatus(ContainerStatusRequest) returns (Contain
erStatusResponse) {}
    // UpdateContainerResources updates ContainerConfig of the c
ontainer.
    rpc UpdateContainerResources(UpdateContainerResourcesRequest)
 returns (UpdateContainerResourcesResponse) {}

    // ExecSync runs a command in a container synchronously.
    rpc ExecSync(ExecSyncRequest) returns (ExecSyncResponse) {}
    // Exec prepares a streaming endpoint to execute a command i
n the container.
    rpc Exec(ExecRequest) returns (ExecResponse) {}
    // Attach prepares a streaming endpoint to attach to a runni
ng container.
    rpc Attach(AttachRequest) returns (AttachResponse) {}
    // PortForward prepares a streaming endpoint to forward port
s from a PodSandbox.
    rpc PortForward(PortForwardRequest) returns (PortForwardResp
onse) {}

    // ContainerStats returns stats of the container. If the con
tainer does not
```

```
    // exist, the call returns an error.
    rpc ContainerStats(ContainerStatsRequest) returns (Container
StatsResponse) {}
    // ListContainerStats returns stats of all running container
s.
    rpc ListContainerStats(ListContainerStatsRequest) returns (L
istContainerStatsResponse) {}

    // UpdateRuntimeConfig updates the runtime configuration bas
ed on the given request.
    rpc UpdateRuntimeConfig(UpdateRuntimeConfigRequest) returns (
UpdateRuntimeConfigResponse) {}

    // Status returns the status of the runtime.
    rpc Status(StatusRequest) returns (StatusResponse) {}
}

// ImageService defines the public APIs for managing images.
service ImageService {
    // ListImages lists existing images.
    rpc ListImages(ListImagesRequest) returns (ListImagesResponse
) {}
    // ImageStatus returns the status of the image. If the image
 is not
    // present, returns a response with ImageStatusResponse.Imag
e set to
    // nil.
    rpc ImageStatus(ImageStatusRequest) returns (ImageStatusResp
onse) {}
    // PullImage pulls an image with authentication config.
    rpc PullImage(PullImageRequest) returns (PullImageResponse) {
}
    // RemoveImage removes the image.
    // This call is idempotent, and must not return an error if
the image has
    // already been removed.
    rpc RemoveImage(RemoveImageRequest) returns (RemoveImageResp
onse) {}
    // ImageFSInfo returns information of the filesystem that is
 used to store images.
    rpc ImageFsInfo(ImageFsInfoRequest) returns (ImageFsInfoResp
onse) {}
}
```

这其中包含了两个gRPC服务：

- **RuntimeService**：容器和Sandbox运行时管理
- **ImageService**：提供了从镜像仓库拉取、查看、和移除镜像的RPC。

当前支持的CRI后端

我们最初在使用Kubernetes时通常会默认使用Docker作为容器运行时，其实从Kubernetes1.5开始已经开始支持CRI，目前是处于Alpha版本，通过CRI接口可以指定使用其它容器运行时作为Pod的后端，目前支持 CRI 的后端有：

- cri-o：同时兼容OCI和CRI的容器运行时
- cri-containerd：基于Containerd的Kubernetes CRI 实现
- rkt：由于CoreOS主推的用来跟docker抗衡的容器运行时
- frakti：基于hypervisor的CRI
- docker：kuberentes最初就开始支持的容器运行时，目前还没完全从kubelet中解耦，docker公司同时推广了OCI标准
- clear-containers：由Intel推出的同时兼容OCI和CRI的容器运行时
- kata-containers：符合OCI规范同时兼容CRI

CRI是由SIG-Node来维护的。

参考

- Kubernetes CRI and Minikube
- Kubernetes container runtime interface
- CRI-O and Alternative Runtimes in Kubernetes
- Docker、Containerd、RunC...：你应该知道的所有
- Introducing Container Runtime Interface (CRI) in Kubernetes

CNI - Container Network Interface（容器网络接口）

CNI（Container Network Interface）是CNCF旗下的一个项目，由一组用于配置Linux容器的网络接口的规范和库组成，同时还包含了一些插件。CNI仅关心容器创建时的网络分配，和当容器被删除时释放网络资源。通过此链接浏览该项目：https://github.com/containernetworking/cni。

Kubernetes源码的 `vendor/github.com/containernetworking/cni/libcni` 目录中已经包含了CNI的代码，也就是说kubernetes中已经内置了CNI。

接口定义

CNI的接口中包括以下几个方法：

```
type CNI interface {
    AddNetworkList(net *NetworkConfigList, rt *RuntimeConf) (types.Result, error)
    DelNetworkList(net *NetworkConfigList, rt *RuntimeConf) error

    AddNetwork(net *NetworkConfig, rt *RuntimeConf) (types.Result, error)
    DelNetwork(net *NetworkConfig, rt *RuntimeConf) error
}
```

该接口只有四个方法，添加网络、删除网络、添加网络列表、删除网络列表。

设计考量

CNI设计的时候考虑了以下问题：

- 容器运行时必须在调用任何插件之前为容器创建一个新的网络命名空间。
- 然后，运行时必须确定这个容器应属于哪个网络，并为每个网络确定哪些插件必须被执行。
- 网络配置采用JSON格式，可以很容易地存储在文件中。网络配置包括必填字

段，如 `name` 和 `type` 以及插件（类型）。网络配置允许字段在调用之间改变值。为此，有一个可选的字段 `args` ，必须包含不同的信息。
- 容器运行时必须按顺序为每个网络执行相应的插件，将容器添加到每个网络中。
- 在完成容器生命周期后，运行时必须以相反的顺序执行插件（相对于执行添加容器的顺序）以将容器与网络断开连接。
- 容器运行时不能为同一容器调用并行操作，但可以为不同的容器调用并行操作。
- 容器运行时必须为容器订阅ADD和DEL操作，这样ADD后面总是跟着相应的DEL。 DEL可能跟着额外的DEL，但是，插件应该允许处理多个DEL（即插件DEL应该是幂等的）。
- 容器必须由ContainerID唯一标识。存储状态的插件应该使用（网络名称，容器ID）的主键来完成。
- 运行时不能调用同一个网络名称或容器ID执行两次ADD（没有相应的DEL）。换句话说，给定的容器ID必须只能添加到特定的网络一次。

CNI插件

CNI插件必须实现一个可执行文件，这个文件可以被容器管理系统（例如rkt或Kubernetes）调用。

CNI插件负责将网络接口插入容器网络命名空间（例如，veth对的一端），并在主机上进行任何必要的改变（例如将veth的另一端连接到网桥）。然后将IP分配给接口，并通过调用适当的IPAM插件来设置与“IP地址管理”部分一致的路由。

参数

CNI插件必须支持以下操作：

将容器添加到网络

参数：

- 版本。调用者正在使用的CNI规范（容器管理系统或调用插件）的版本。
- 容器**ID** 。由运行时分配的容器的唯一明文标识符。一定不能是空的。
- 网络命名空间路径。要添加的网络名称空间的路径，即 `/proc/[pid]/ns/net` 或绑定挂载/链接。

- 网络配置。描述容器可以加入的网络的JSON文档。架构如下所述。
- 额外的参数。这提供了一个替代机制，允许在每个容器上简单配置CNI插件。
- 容器内接口的名称。这是应该分配给容器（网络命名空间）内创建的接口的名称；因此它必须符合Linux接口名称上的标准限制。

结果：

- 接口列表。根据插件的不同，这可以包括沙箱（例如容器或管理程序）接口名称和/或主机接口名称，每个接口的硬件地址以及接口所在的沙箱（如果有的话）的详细信息。
- 分配给每个接口的IP配置。分配给沙箱和/或主机接口的IPv4和/或IPv6地址，网关和路由。
- DNS信息。包含nameserver、domain、search domain和option的DNS信息的字典。

从网络中删除容器

参数：

- 版本。调用者正在使用的CNI规范（容器管理系统或调用插件）的版本。
- 容器ID ，如上所述。
- 网络命名空间路径，如上定义。
- 网络配置，如上所述。
- 额外的参数，如上所述。
- 上面定义的容器内的接口的名称。

- 所有参数应与传递给相应的添加操作的参数相同。
- 删除操作应释放配置的网络中提供的containerid拥有的所有资源。

报告版本

- 参数：无。
- 结果：插件支持的CNI规范版本信息。

```
{
“cniVersion”：“0.3.1”，//此输出使用的CNI规范的版本
“supportedVersions”：[“0.1.0”，“0.2.0”，“0.3.0”，“0.3.1”] //此插件支持的CNI规范版本列表
}
```

CNI插件的详细说明请参考：CNI SPEC。

IP分配

作为容器网络管理的一部分，CNI插件需要为接口分配（并维护）IP地址，并安装与该接口相关的所有必要路由。这给了CNI插件很大的灵活性，但也给它带来了很大的负担。众多的CNI插件需要编写相同的代码来支持用户需要的多种IP管理方案（例如dhcp、host-local）。

为了减轻负担，使IP管理策略与CNI插件类型解耦，我们定义了IP地址管理插件（IPAM插件）。CNI插件的职责是在执行时恰当地调用IPAM插件。 IPAM插件必须确定接口IP/subnet，网关和路由，并将此信息返回到“主”插件来应用配置。 IPAM插件可以通过协议（例如dhcp）、存储在本地文件系统上的数据、网络配置文件的“ipam”部分或上述的组合来获得信息。

IPAM插件

像CNI插件一样，调用IPAM插件的可执行文件。可执行文件位于预定义的路径列表中，通过 `CNI_PATH` 指示给CNI插件。 IPAM插件必须接收所有传入CNI插件的相同环境变量。就像CNI插件一样，IPAM插件通过stdin接收网络配置。

可用插件

Main：接口创建

- **bridge**：创建网桥，并添加主机和容器到该网桥
- **ipvlan**：在容器中添加一个ipvlan接口
- **loopback**：创建一个回环接口
- **macvlan**：创建一个新的MAC地址，将所有的流量转发到容器
- **ptp**：创建veth对
- **vlan**：分配一个vlan设备

IPAM：IP地址分配

- **dhcp**：在主机上运行守护程序，代表容器发出DHCP请求
- **host-local**：维护分配IP的本地数据库

Meta：其它插件

- **flannel**：根据flannel的配置文件创建接口
- **tuning**：调整现有接口的sysctl参数
- **portmap**：一个基于iptables的portmapping插件。将端口从主机的地址空间映射到容器。

参考

- https://github.com/containernetworking/cni
- https://github.com/containernetworking/plugins
- Container Networking Interface Specification
- CNI Extension conventions

CSI - Container Storage Interface（容器存储接口）

CSI 代表容器存储接口，CSI 试图建立一个行业标准接口的规范，借助 CSI 容器编排系统（CO）可以将任意存储系统暴露给自己的容器工作负载。有关详细信息，请查看设计方案。

`csi` 卷类型是一种 in-tree（即跟其它存储插件在同一个代码路径下，随 Kubernetes 的代码同时编译的） 的 CSI 卷插件，用于 Pod 与在同一节点上运行的外部 CSI 卷驱动程序交互。部署 CSI 兼容卷驱动后，用户可以使用 `csi` 作为卷类型来挂载驱动提供的存储。

CSI 持久化卷支持是在 Kubernetes v1.9 中引入的，作为一个 alpha 特性，必须由集群管理员明确启用。换句话说，集群管理员需要在 apiserver、controller-manager 和 kubelet 组件的“ `--feature-gates =` ”标志中加上“ `CSIPersistentVolume = true` ”。

CSI 持久化卷具有以下字段可供用户指定：

- `driver` ：一个字符串值，指定要使用的卷驱动程序的名称。必须少于 63 个字符，并以一个字符开头。驱动程序名称可以包含“。”、“-”、“_”或数字。
- `volumeHandle` ：一个字符串值，唯一标识从 CSI 卷插件的 `CreateVolume` 调用返回的卷名。随后在卷驱动程序的所有后续调用中使用卷句柄来引用该卷。
- `readOnly` ：一个可选的布尔值，指示卷是否被发布为只读。默认是 false。

使用说明

下面将介绍如何使用 CSI。

动态配置

可以通过为 CSI 创建插件 `StorageClass` 来支持动态配置的 CSI Storage 插件启用自动创建/删除 。

例如，以下 `StorageClass` 允许通过名为 `com.example.team/csi-driver` 的 CSI Volume Plugin 动态创建 “fast-storage” Volume。

```yaml
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: fast-storage
provisioner: com.example.team/csi-driver
parameters:
  type: pd-ssd
```

要触发动态配置，请创建一个 `PersistentVolumeClaim` 对象。例如，下面的 PersistentVolumeClaim 可以使用上面的 StorageClass 触发动态配置。

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: my-request-for-storage
spec:
  accessModes:
  - ReadWriteOnce
  resources:
    requests:
      storage: 5Gi
  storageClassName: fast-storage
```

当动态创建 Volume 时，通过 CreateVolume 调用，将参数 `type：pd-ssd` 传递给 CSI 插件 `com.example.team/csi-driver` 。作为响应，外部 Volume 插件会创建一个新 Volume，然后自动创建一个 `PersistentVolume` 对象来对应前面的 PVC 。然后，Kubernetes 会将新的 `PersistentVolume` 对象绑定到 `PersistentVolumeClaim` ，使其可以使用。

如果 `fast-storage` StorageClass 被标记为默认值，则不需要在 `PersistentVolumeClaim` 中包含 StorageClassName，它将被默认使用。

预配置 Volume

您可以通过手动创建一个 `PersistentVolume` 对象来展示现有 Volumes，从而在 Kubernetes 中暴露预先存在的 Volume。例如，暴露属于 `com.example.team/csi-driver` 这个 CSI 插件的 `existingVolumeName` `Volume` ：

```yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: my-manually-created-pv
spec:
  capacity:
    storage: 5Gi
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Retain
  csi:
    driver: com.example.team/csi-driver
    volumeHandle: existingVolumeName
    readOnly: false
```

附着和挂载

您可以在任何的 pod 或者 pod 的 template 中引用绑定到 CSI volume 上的 `PersistentVolumeClaim`。

```yaml
kind: Pod
apiVersion: v1
metadata:
  name: my-pod
spec:
  containers:
    - name: my-frontend
      image: dockerfile/nginx
      volumeMounts:
      - mountPath: "/var/www/html"
        name: my-csi-volume
  volumes:
    - name: my-csi-volume
      persistentVolumeClaim:
        claimName: my-request-for-storage
```

当一个引用了 CSI Volume 的 pod 被调度时， Kubernetes 将针对外部 CSI 插件进行相应的操作，以确保特定的 Volume 被 attached、mounted， 并且能被 pod 中的容器使用。

关于 CSI 实现的详细信息请参考设计文档。

创建 CSI 驱动

Kubernetes 尽可能少地指定 CSI Volume 驱动程序的打包和部署规范。这里记录了在 Kubernetes 上部署 CSI Volume 驱动程序的最低要求。

最低要求文件还包含概述部分，提供了在 Kubernetes 上部署任意容器化 CSI 驱动程序的建议机制。存储提供商可以运用这个机制来简化 Kubernetes 上容器式 CSI 兼容 Volume 驱动程序的部署。

作为推荐部署的一部分，Kubernetes 团队提供以下 sidecar（辅助）容器：

- External-attacher

  可监听 Kubernetes VolumeAttachment 对象并触发 ControllerPublish 和 ControllerUnPublish 操作的 sidecar 容器，通过 CSI endpoint 触发 ；

- External-provisioner

  监听 Kubernetes PersistentVolumeClaim 对象的 sidecar 容器，并触发对 CSI 端点的 CreateVolume 和DeleteVolume 操作；

- Driver-registrar

  使用 Kubelet（将来）注册 CSI 驱动程序的 sidecar 容器，并将 `NodeId`（通过 `GetNodeID` 调用检索到 CSI endpoint）添加到 Kubernetes Node API 对象的 annotation 里面。

存储供应商完全可以使用这些组件来为其插件构建 Kubernetes Deployment，同时让它们的 CSI 驱动程序完全意识不到 Kubernetes 的存在。

另外 CSI 驱动完全是由第三方存储供应商自己维护的，在 kubernetes 1.9 版本中 CSI 还处于 alpha 版本。

参考

- Introducing Container Storage Interface (CSI) Alpha for Kubernetes
- Container Storage Interface (CSI)

Kubernetes中的网络解析——以flannel为例

我们当初使用kubernetes-vagrant-centos-cluster安装了拥有三个节点的kubernetes集群，节点的状态如下所述。

```
[root@node1 ~]# kubectl get nodes -o wide
NAME      STATUS    ROLES     AGE       VERSION   EXTERNAL-IP
OS-IMAGE                KERNEL-VERSION              CONTAINER-R
UNTIME
node1     Ready     <none>    2d        v1.9.1    <none>
CentOS Linux 7 (Core)   3.10.0-693.11.6.el7.x86_64   docker://1.
12.6
node2     Ready     <none>    2d        v1.9.1    <none>
CentOS Linux 7 (Core)   3.10.0-693.11.6.el7.x86_64   docker://1.
12.6
node3     Ready     <none>    2d        v1.9.1    <none>
CentOS Linux 7 (Core)   3.10.0-693.11.6.el7.x86_64   docker://1.
12.6
```

当前Kubernetes集群中运行的所有Pod信息：

```
[root@node1 ~]# kubectl get pods --all-namespaces -o wide
NAMESPACE     NAME
READY     STATUS    RESTARTS   AGE       IP             NODE
kube-system   coredns-5984fb8cbb-sjqv9
1/1       Running   0          1h        172.33.68.2    node1
kube-system   coredns-5984fb8cbb-tkfrc
1/1       Running   1          1h        172.33.96.3    node3
kube-system   heapster-v1.5.0-684c7f9488-z6sdz
4/4       Running   0          1h        172.33.31.3    node2
kube-system   kubernetes-dashboard-6b66b8b96c-mnm2c
1/1       Running   0          1h        172.33.31.2    node2
kube-system   monitoring-influxdb-grafana-v4-54b7854697-tw9cd
2/2       Running   2          1h        172.33.96.2    node3
```

当前etcd中的注册的宿主机的pod地址网段信息：

```
[root@node1 ~]# etcdctl ls /kube-centos/network/subnets
/kube-centos/network/subnets/172.33.68.0-24
/kube-centos/network/subnets/172.33.31.0-24
/kube-centos/network/subnets/172.33.96.0-24
```

而每个node上的Pod子网是根据我们在安装flannel时配置来划分的，在etcd中查看该配置：

```
[root@node1 ~]# etcdctl get /kube-centos/network/config
{"Network":"172.33.0.0/16","SubnetLen":24,"Backend":{"Type":"hos
t-gw"}}
```

我们知道Kubernetes集群内部存在三类IP，分别是：

- Node IP：宿主机的IP地址
- Pod IP：使用网络插件创建的IP（如flannel），使跨主机的Pod可以互通
- Cluster IP：虚拟IP，通过iptables规则访问服务

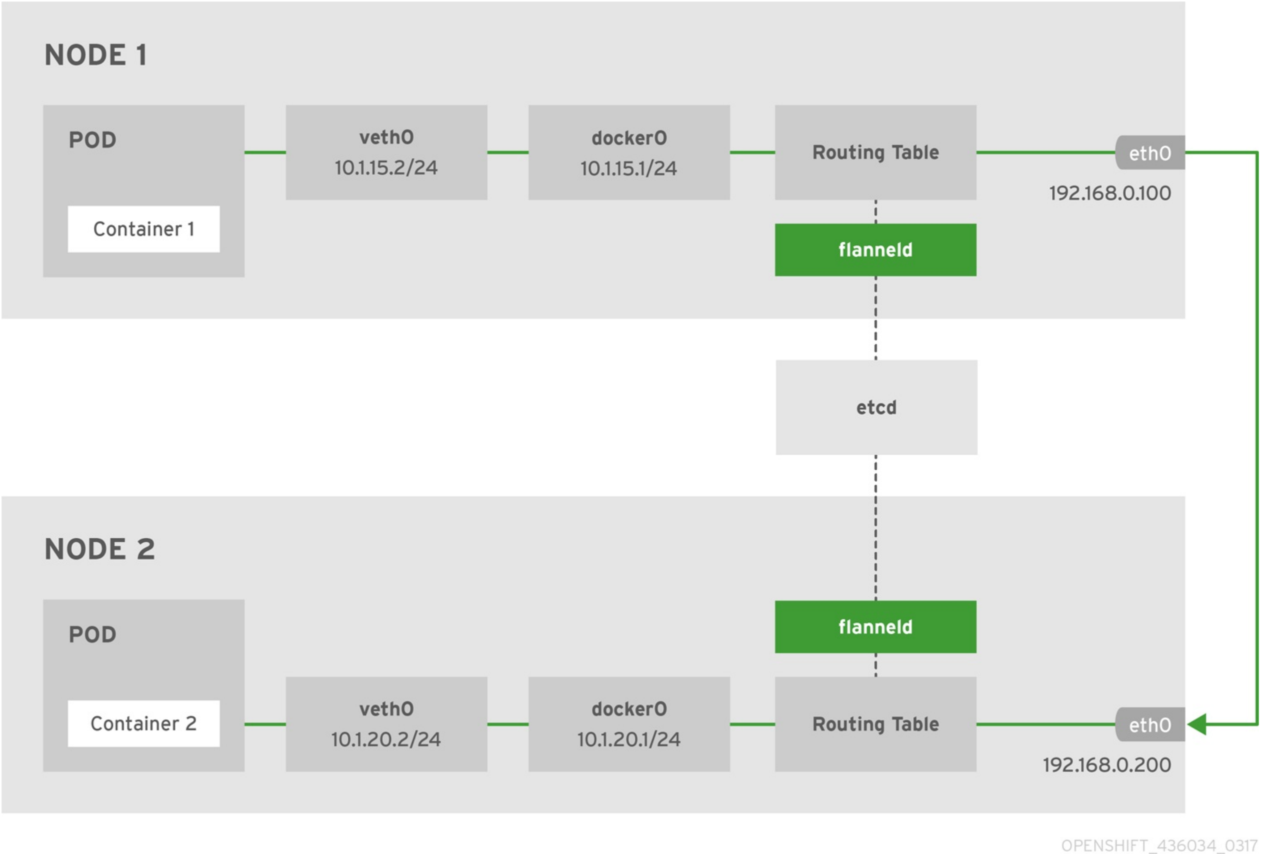
在安装node节点的时候，节点上的进程是按照flannel -> docker -> kubelet -> kube-proxy的顺序启动的，我们下面也会按照该顺序来讲解，flannel的网络划分和如何与docker交互，如何通过iptables访问service。

Flannel

Flannel是作为一个二进制文件的方式部署在每个node上，主要实现两个功能：

- 为每个node分配subnet，容器将自动从该子网中获取IP地址
- 当有node加入到网络中时，为每个node增加路由配置

下面是使用 `host-gw` backend的flannel网络架构图：



图片 - *flannel*网络架构（图片来自*openshift*）

注意：以上IP非本示例中的IP，但是不影响读者理解。

Node1上的flannel配置如下:

```
[root@node1 ~]# cat /usr/lib/systemd/system/flanneld.service
[Unit]
Description=Flanneld overlay address etcd agent
After=network.target
After=network-online.target
Wants=network-online.target
After=etcd.service
Before=docker.service

[Service]
Type=notify
EnvironmentFile=/etc/sysconfig/flanneld
EnvironmentFile=-/etc/sysconfig/docker-network
ExecStart=/usr/bin/flanneld-start $FLANNEL_OPTIONS
ExecStartPost=/usr/libexec/flannel/mk-docker-opts.sh -k DOCKER_N
ETWORK_OPTIONS -d /run/flannel/docker
Restart=on-failure

[Install]
WantedBy=multi-user.target
RequiredBy=docker.service
```

其中有两个环境变量文件的配置如下：

```
[root@node1 ~]# cat /etc/sysconfig/flanneld
# Flanneld configuration options
FLANNEL_ETCD_ENDPOINTS="http://172.17.8.101:2379"
FLANNEL_ETCD_PREFIX="/kube-centos/network"
FLANNEL_OPTIONS="-iface=eth2"
```

上面的配置文件仅供flanneld使用。

```
[root@node1 ~]# cat /etc/sysconfig/docker-network
# /etc/sysconfig/docker-network
DOCKER_NETWORK_OPTIONS=
```

还有一个 `ExecStartPost=/usr/libexec/flannel/mk-docker-opts.sh -k DOCKER_NETWORK_OPTIONS -d /run/flannel/docker` ，其中的 `/usr/libexec/flannel/mk-docker-opts.sh` 脚本是在flanneld启动后运行，将会生成两个环境变量配置文件：

- /run/flannel/docker
- /run/flannel/subnet.env

我们再来看下 `/run/flannel/docker` 的配置。

```
[root@node1 ~]# cat /run/flannel/docker
DOCKER_OPT_BIP="--bip=172.33.68.1/24"
DOCKER_OPT_IPMASQ="--ip-masq=true"
DOCKER_OPT_MTU="--mtu=1500"
DOCKER_NETWORK_OPTIONS=" --bip=172.33.68.1/24 --ip-masq=true --m
tu=1500"
```

如果你使用 `systemctl` 命令先启动flannel后启动docker的话，docker将会读取以上环境变量。

我们再来看下 `/run/flannel/subnet.env` 的配置。

```
[root@node1 ~]# cat /run/flannel/subnet.env
FLANNEL_NETWORK=172.33.0.0/16
FLANNEL_SUBNET=172.33.68.1/24
FLANNEL_MTU=1500
FLANNEL_IPMASQ=false
```

以上环境变量是flannel向etcd中注册的。

Docker

Node1的docker配置如下：

```
[root@node1 ~]# cat /usr/lib/systemd/system/docker.service
[Unit]
Description=Docker Application Container Engine
Documentation=http://docs.docker.com
After=network.target rhel-push-plugin.socket registries.service
Wants=docker-storage-setup.service
Requires=docker-cleanup.timer

[Service]
Type=notify
NotifyAccess=all
EnvironmentFile=-/run/containers/registries.conf
EnvironmentFile=-/etc/sysconfig/docker
EnvironmentFile=-/etc/sysconfig/docker-storage
EnvironmentFile=-/etc/sysconfig/docker-network
Environment=GOTRACEBACK=crash
Environment=DOCKER_HTTP_HOST_COMPAT=1
Environment=PATH=/usr/libexec/docker:/usr/bin:/usr/sbin
ExecStart=/usr/bin/dockerd-current \
          --add-runtime docker-runc=/usr/libexec/docker/docker-r
unc-current \
          --default-runtime=docker-runc \
          --exec-opt native.cgroupdriver=systemd \
          --userland-proxy-path=/usr/libexec/docker/docker-proxy
-current \
          $OPTIONS \
          $DOCKER_STORAGE_OPTIONS \
          $DOCKER_NETWORK_OPTIONS \
          $ADD_REGISTRY \
          $BLOCK_REGISTRY \
          $INSECURE_REGISTRY\
          $REGISTRIES
ExecReload=/bin/kill -s HUP $MAINPID
LimitNOFILE=1048576
LimitNPROC=1048576
LimitCORE=infinity
TimeoutStartSec=0
Restart=on-abnormal
MountFlags=slave
KillMode=process

[Install]
WantedBy=multi-user.target
```

查看Node1上的docker启动参数：

```
[root@node1 ~]# systemctl status -l docker
● docker.service - Docker Application Container Engine
   Loaded: loaded (/usr/lib/systemd/system/docker.service; enabl
ed; vendor preset: disabled)
  Drop-In: /usr/lib/systemd/system/docker.service.d
           └─flannel.conf
   Active: active (running) since Fri 2018-02-02 22:52:43 CST; 2
h 28min ago
     Docs: http://docs.docker.com
 Main PID: 4334 (dockerd-current)
   CGroup: /system.slice/docker.service
           ▸ 4334 /usr/bin/dockerd-current --add-runtime docker-
runc=/usr/libexec/docker/docker-runc-current --default-runtime=d
ocker-runc --exec-opt native.cgroupdriver=systemd --userland-pro
xy-path=/usr/libexec/docker/docker-proxy-current --selinux-enabl
ed --log-driver=journald --signature-verification=false --bip=17
2.33.68.1/24 --ip-masq=true --mtu=1500
```

我们可以看到在docker在启动时有如下参数： `--bip=172.33.68.1/24 --ip-masq=true --mtu=1500` 。上述参数flannel启动时运行的脚本生成的，通过环境变量传递过来的。

我们查看下node1宿主机上的网络接口：

```
[root@node1 ~]# ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKN
OWN qlen 1
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_
fast state UP qlen 1000
    link/ether 52:54:00:00:57:32 brd ff:ff:ff:ff:ff:ff
    inet 10.0.2.15/24 brd 10.0.2.255 scope global dynamic eth0
       valid_lft 85095sec preferred_lft 85095sec
    inet6 fe80::5054:ff:fe00:5732/64 scope link
       valid_lft forever preferred_lft forever
3: eth1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_
fast state UP qlen 1000
    link/ether 08:00:27:7b:0f:b1 brd ff:ff:ff:ff:ff:ff
    inet 172.17.8.101/24 brd 172.17.8.255 scope global eth1
       valid_lft forever preferred_lft forever
4: eth2: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_
fast state UP qlen 1000
    link/ether 08:00:27:ef:25:06 brd ff:ff:ff:ff:ff:ff
    inet 172.30.113.231/21 brd 172.30.119.255 scope global dynam
ic eth2
       valid_lft 85096sec preferred_lft 85096sec
    inet6 fe80::a00:27ff:feef:2506/64 scope link
       valid_lft forever preferred_lft forever
5: docker0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noq
ueue state UP
    link/ether 02:42:d0:ae:80:ea brd ff:ff:ff:ff:ff:ff
    inet 172.33.68.1/24 scope global docker0
       valid_lft forever preferred_lft forever
    inet6 fe80::42:d0ff:feae:80ea/64 scope link
       valid_lft forever preferred_lft forever
7: veth295bef2@if6: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 q
disc noqueue master docker0 state UP
    link/ether 6a:72:d7:9f:29:19 brd ff:ff:ff:ff:ff:ff link-netn
sid 0
    inet6 fe80::6872:d7ff:fe9f:2919/64 scope link
       valid_lft forever preferred_lft forever
```

我们分类来解释下该虚拟机中的网络接口。

- lo：回环网络，127.0.0.1
- eth0：NAT网络，虚拟机创建时自动分配，仅可以在几台虚拟机之间访问
- eth1：bridge网络，使用vagrant分配给虚拟机的地址，虚拟机之间和本地电脑都可以访问
- eth2：bridge网络，使用DHCP分配，用于访问互联网的网卡

- docker0：bridge网络，docker默认使用的网卡，作为该节点上所有容器的虚拟交换机
- veth295bef2@if6：veth pair，连接docker0和Pod中的容器。veth pair可以理解为使用网线连接好的两个接口，把两个端口放到两个namespace中，那么这两个namespace就能打通。参考linux 网络虚拟化： network namespace 简介。

我们再看下该节点的docker上有哪些网络。

```
[root@node1 ~]# docker network ls
NETWORK ID          NAME                DRIVER              SCOP
E
940bb75e653b        bridge              bridge              loca
l
d94c046e105d        host                host                loca
l
2db7597fd546        none                null                loca
l
```

再检查下bridge网络 `940bb75e653b` 的信息。

```
[root@node1 ~]# docker network inspect 940bb75e653b
[
    {
        "Name": "bridge",
        "Id": "940bb75e653bfa10dab4cce8813c2b3ce17501e4e4935f7dc
13805a61b732d2c",
        "Scope": "local",
        "Driver": "bridge",
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "default",
            "Options": null,
            "Config": [
                {
                    "Subnet": "172.33.68.1/24",
                    "Gateway": "172.33.68.1"
                }
            ]
        },
        "Internal": false,
        "Containers": {
            "944d4aa660e30e1be9a18d30c9dcfa3b0504d1e5dbd00f3004b
76582f1c9a85b": {
                "Name": "k8s_POD_coredns-5984fb8cbb-sjqv9_kube-s
ystem_c5a2e959-082a-11e8-b4cd-525400005732_0",
                "EndpointID": "7397d7282e464fc4ec5756d6b328df889
cdf46134dbbe3753517e175d3844a85",
                "MacAddress": "02:42:ac:21:44:02",
                "IPv4Address": "172.33.68.2/24",
                "IPv6Address": ""
            }
        },
        "Options": {
            "com.docker.network.bridge.default_bridge": "true",
            "com.docker.network.bridge.enable_icc": "true",
            "com.docker.network.bridge.enable_ip_masquerade": "t
rue",
            "com.docker.network.bridge.host_binding_ipv4": "0.0.
0.0",
            "com.docker.network.bridge.name": "docker0",
            "com.docker.network.driver.mtu": "1500"
        },
        "Labels": {}
    }
]
```

我们可以看到该网络中的 `Config` 与docker的启动配置相符。

Node1上运行的容器：

```
[root@node1 ~]# docker ps
CONTAINER ID        IMAGE
                                                                    COMMAND
                 CREATED             STATUS                         PORTS
             NAMES
a37407a234dd        docker.io/coredns/coredns@sha256:adf2e5b4504
ef9ffa43f16010bd064273338759e92f6f616dd159115748799bc   "/coredn
s -conf /etc/"   About an hour ago   Up About an hour
             k8s_coredns_coredns-5984fb8cbb-sjqv9_kube-system_c5
a2e959-082a-11e8-b4cd-525400005732_0
944d4aa660e3        docker.io/openshift/origin-pod
                                                                "/usr/bi
n/pod"           About an hour ago   Up About an hour
             k8s_POD_coredns-5984fb8cbb-sjqv9_kube-system_c5a2e9
59-082a-11e8-b4cd-525400005732_0
```

我们可以看到当前已经有2个容器在运行。

Node1上的路由信息：

```
[root@node1 ~]# route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref
    Use Iface
0.0.0.0         10.0.2.2        0.0.0.0         UG    100    0
      0 eth0
0.0.0.0         172.30.116.1    0.0.0.0         UG    101    0
      0 eth2
10.0.2.0        0.0.0.0         255.255.255.0   U     100    0
      0 eth0
172.17.8.0      0.0.0.0         255.255.255.0   U     100    0
      0 eth1
172.30.112.0    0.0.0.0         255.255.248.0   U     100    0
      0 eth2
172.33.68.0     0.0.0.0         255.255.255.0   U     0      0
      0 docker0
172.33.96.0     172.30.118.65   255.255.255.0   UG    0      0
      0 eth2
```

以上路由信息是由flannel添加的，当有新的节点加入到Kubernetes集群中后，每个节点上的路由表都将增加。

我们在node上来 `traceroute` 下node3上的 `coredns-5984fb8cbb-tkfrc` 容器，其IP地址是 `172.33.96.3` ，看看其路由信息。

```
[root@node1 ~]# traceroute 172.33.96.3
traceroute to 172.33.96.3 (172.33.96.3), 30 hops max, 60 byte pa
ckets
 1  172.30.118.65 (172.30.118.65)  0.518 ms  0.367 ms  0.398 ms
 2  172.33.96.3 (172.33.96.3)  0.451 ms  0.352 ms  0.223 ms
```

我们看到路由直接经过node3的公网IP后就到达了node3节点上的Pod。

Node1的iptables信息：

```
[root@node1 ~]# iptables -L
Chain INPUT (policy ACCEPT)
target     prot opt source               destination
KUBE-FIREWALL  all  --  anywhere             anywhere
KUBE-SERVICES  all  --  anywhere             anywhere
  /* kubernetes service portals */

Chain FORWARD (policy ACCEPT)
target     prot opt source               destination
KUBE-FORWARD  all  --  anywhere             anywhere
 /* kubernetes forward rules */
DOCKER-ISOLATION  all  --  anywhere             anywhere
DOCKER     all  --  anywhere             anywhere
ACCEPT     all  --  anywhere             anywhere             ct
state RELATED,ESTABLISHED
ACCEPT     all  --  anywhere             anywhere
ACCEPT     all  --  anywhere             anywhere

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
KUBE-FIREWALL  all  --  anywhere             anywhere
KUBE-SERVICES  all  --  anywhere             anywhere
  /* kubernetes service portals */

Chain DOCKER (1 references)
target     prot opt source               destination

Chain DOCKER-ISOLATION (1 references)
target     prot opt source               destination
RETURN     all  --  anywhere             anywhere

Chain KUBE-FIREWALL (2 references)
target     prot opt source               destination
DROP       all  --  anywhere             anywhere             /*
 kubernetes firewall for dropping marked packets */ mark match 0
x8000/0x8000

Chain KUBE-FORWARD (1 references)
target     prot opt source               destination
ACCEPT     all  --  anywhere             anywhere             /*
 kubernetes forwarding rules */ mark match 0x4000/0x4000
ACCEPT     all  --  10.254.0.0/16        anywhere             /*
 kubernetes forwarding conntrack pod source rule */ ctstate RELA
TED,ESTABLISHED
ACCEPT     all  --  anywhere             10.254.0.0/16        /*
 kubernetes forwarding conntrack pod destination rule */ ctstate
 RELATED,ESTABLISHED

Chain KUBE-SERVICES (2 references)
target     prot opt source               destination
```

从上面的iptables中可以看到注入了很多Kuberentes service的规则，请参考 iptables 规则获取更多详细信息。

参考

- coreos/flannel - github.com
- linux 网络虚拟化： network namespace 简介
- Linux虚拟网络设备之veth
- iptables 规则
- flannel host-gw network
- flannel - openshift.com

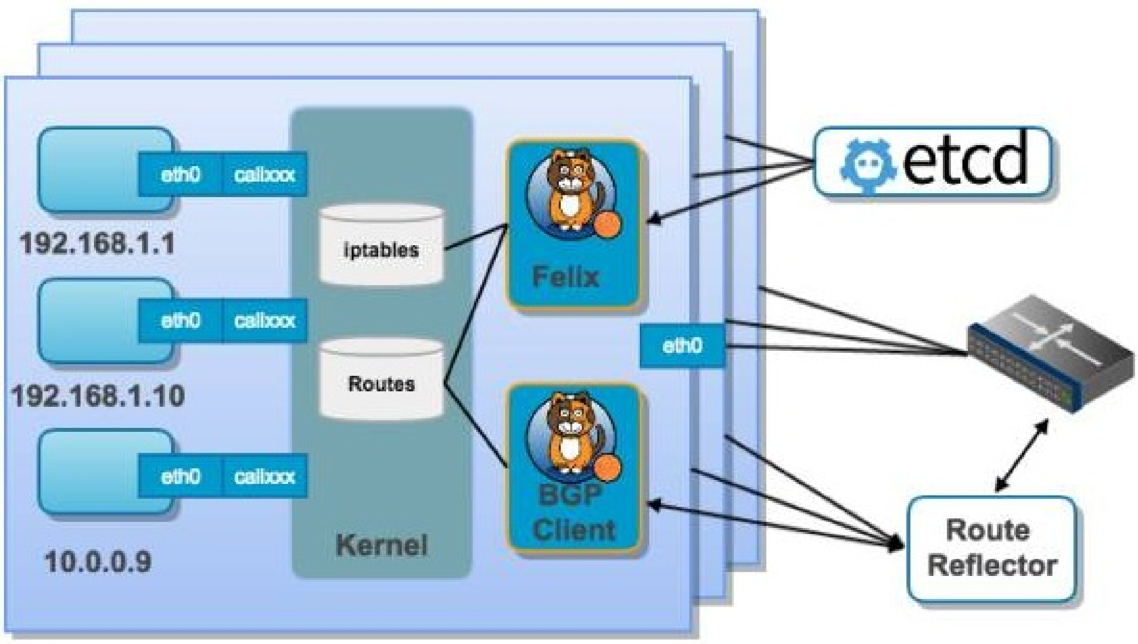
Kubernetes中的网络解析——以calico为例

概念

Calico 创建和管理一个扁平的三层网络(不需要overlay)，每个容器会分配一个可路由的ip。由于通信时不需要解包和封包，网络性能损耗小，易于排查，且易于水平扩展。

小规模部署时可以通过bgp client直接互联，大规模下可通过指定的BGP route reflector来完成，这样保证所有的数据流量都是通过IP路由的方式完成互联的。Calico基于iptables还提供了丰富而灵活的网络Policy，保证通过各个节点上的ACLs来提供Workload的多租户隔离、安全组以及其他可达性限制等功能。

Calico架构



图片 - *CRI*架构-图片来自*https://www.jianshu.com/p/f0177b84de66*

calico主要由Felix,etcd,BGP client,BGP Route Reflector组成。

- Etcd：负责存储网络信息

- BGP client：负责将Felix配置的路由信息分发到其他节点
- Felix：Calico Agent，每个节点都需要运行，主要负责配置路由、配置ACLs、报告状态
- BGP Route Reflector：大规模部署时需要用到，作为BGP client的中心连接点，可以避免每个节点互联

部署

```
mkdir /etc/cni/net.d/

kubectl apply -f https://docs.projectcalico.org/v3.0/getting-sta
rted/kubernetes/installation/rbac.yaml

wget https://docs.projectcalico.org/v3.0/getting-started/kuberne
tes/installation/hosted/calico.yaml

修改etcd_endpoints的值和默认的192.168.0.0/16(不能和已有网段冲突)

kubectl apply -f calico.yaml

wget  https://github.com/projectcalico/calicoctl/releases/downlo
ad/v2.0.0/calicoctl

mv calicoctl /usr/loca/bin && chmod +x /usr/local/bin/calicoctl

calicoctl get ippool

calicoctl get node
```

Objects

以下列举的内容都是 kubernetes 中的 Object，这些对象都可以在 yaml 文件中作为一种 API 类型来配置。

- Pod
- Node
- Namespace
- Service
- Volume
- PersistentVolume
- Deployment
- Secret
- StatefulSet
- DaemonSet
- ServiceAccount
- ReplicationController
- ReplicaSet
- Job
- CronJob
- SecurityContext
- ResourceQuota
- LimitRange
- HorizontalPodAutoscaling
- Ingress
- ConfigMap
- Label
- ThirdPartyResources

我将它们简单的分类为以下几种资源对象：

类别	名称
资源对象	Pod、ReplicaSet、ReplicationController、Deployment、StatefulSet、DaemonSet、Job、CronJob、HorizontalPodAutoscaling
配置对象	Node、Namespace、Service、Secret、ConfigMap、Ingress、Label、ThirdPartyResource、 ServiceAccount
存储对象	Volume、Persistent Volume
策略对象	SecurityContext、ResourceQuota、LimitRange

理解 kubernetes 中的对象

在 Kubernetes 系统中，*Kubernetes* 对象 是持久化的条目。Kubernetes 使用这些条目去表示整个集群的状态。特别地，它们描述了如下信息：

- 什么容器化应用在运行（以及在哪个 Node 上）
- 可以被应用使用的资源
- 关于应用如何表现的策略，比如重启策略、升级策略，以及容错策略

Kubernetes 对象是 "目标性记录" —— 一旦创建对象，Kubernetes 系统将持续工作以确保对象存在。通过创建对象，可以有效地告知 Kubernetes 系统，所需要的集群工作负载看起来是什么样子的，这就是 Kubernetes 集群的 期望状态。

与 Kubernetes 对象工作 —— 是否创建、修改，或者删除 —— 需要使用 Kubernetes API。当使用 `kubectl` 命令行接口时，比如，CLI 会使用必要的 Kubernetes API 调用，也可以在程序中直接使用 Kubernetes API。为了实现该目标，Kubernetes 当前提供了一个 `golang` 客户端库 ，其它语言库（例如 Python）也正在开发中。

对象 Spec 与状态

每个 Kubernetes 对象包含两个嵌套的对象字段，它们负责管理对象的配置：对象 *spec* 和 对象 *status*。*spec* 必须提供，它描述了对象的 期望状态—— 希望对象所具有的特征。*status* 描述了对象的 实际状态，它是由 Kubernetes 系统提供和更新。在任何时刻，Kubernetes 控制平面一直处于活跃状态，管理着对象的实际状态以与我们所期望的状态相匹配。

例如，Kubernetes Deployment 对象能够表示运行在集群中的应用。当创建 Deployment 时，可能需要设置 Deployment 的 spec，以指定该应用需要有 3 个副本在运行。Kubernetes 系统读取 Deployment spec，启动我们所期望的该应用的 3 个实例 —— 更新状态以与 spec 相匹配。如果那些实例中有失败的（一种状态变更），Kubernetes 系统通过修正来响应 spec 和状态之间的不一致 —— 这种情况，启动一个新的实例来替换。

关于对象 spec、status 和 metadata 更多信息，查看 Kubernetes API Conventions。

描述 Kubernetes 对象

当创建 Kubernetes 对象时，必须提供对象的 spec，用来描述该对象的期望状态，以及关于对象的一些基本信息（例如，名称）。当使用 Kubernetes API 创建对象时（或者直接创建，或者基于 `kubectl` ），API 请求必须在请求体中包含 JSON 格式的信息。更常用的是，需要在 **.yaml** 文件中为 **kubectl** 提供这些信息。 `kubectl` 在执行 API 请求时，将这些信息转换成 JSON 格式。

这里有一个 `.yaml` 示例文件，展示了 Kubernetes Deployment 的必需字段和对象 spec：

```yaml
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 3
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
        - containerPort: 80
```

一种创建 Deployment 的方式，类似上面使用 `.yaml` 文件，是使用 `kubectl` 命令行接口（CLI）中的 `kubectl create` 命令，传递 `.yaml` 作为参数。下面是一个示例：

```
$ kubectl create -f docs/user-guide/nginx-deployment.yaml --reco
rd
```

输出类似如下这样：

```
deployment "nginx-deployment" created
```

必需字段

在想要创建的 Kubernetes 对象对应的 `.yaml` 文件中，需要配置如下的字段：

- `apiVersion` - 创建该对象所使用的 Kubernetes API 的版本
- `kind` - 想要创建的对象的类型
- `metadata` - 帮助识别对象唯一性的数据，包括一个 `name` 字符串、UID 和可选的 `namespace`

也需要提供对象的 `spec` 字段。对象 `spec` 的精确格式对每个 Kubernetes 对象来说是不同的，包含了特定于该对象的嵌套字段。Kubernetes API 参考能够帮助我们找到任何我们想创建的对象的 spec 格式。

Pod状态与生命周期管理

该节将带领大家了解Kubernetes中的基本概念，尤其是作为Kubernetes中调度的最基本单位Pod。

本节中包括以下内容：

- 了解Pod的构成
- Pod的生命周期
- Pod中容器的启动顺序模板定义

Kubernetes中的基本组件 `kube-controller-manager` 就是用来控制Pod的状态和生命周期的，在了解各种controller之前我们有必要先了解下Pod本身和其生命周期。

Pod概览

理解Pod

Pod是kubernetes中你可以创建和部署的最小也是最简的单位。一个Pod代表着集群中运行的一个进程。

Pod中封装着应用的容器（有的情况下是好几个容器），存储、独立的网络IP，管理容器如何运行的策略选项。Pod代表着部署的一个单位：kubernetes中应用的一个实例，可能由一个或者多个容器组合在一起共享资源。

> Docker是kubernetes中最常用的容器运行时，但是Pod也支持其他容器运行时。

在Kubrenetes集群中Pod有如下两种使用方式：

- **一个Pod中运行一个容器**。“每个Pod中一个容器”的模式是最常见的用法；在这种使用方式中，你可以把Pod想象成是单个容器的封装，kuberentes管理的是Pod而不是直接管理容器。
- **在一个Pod中同时运行多个容器**。一个Pod中也可以同时封装几个需要紧密耦合互相协作的容器，它们之间共享资源。这些在同一个Pod中的容器可以互相协作成为一个service单位——一个容器共享文件，另一个“sidecar”容器来更新这些文件。Pod将这些容器的存储资源作为一个实体来管理。

Kubernetes Blog 有关于Pod用例的详细信息，查看：

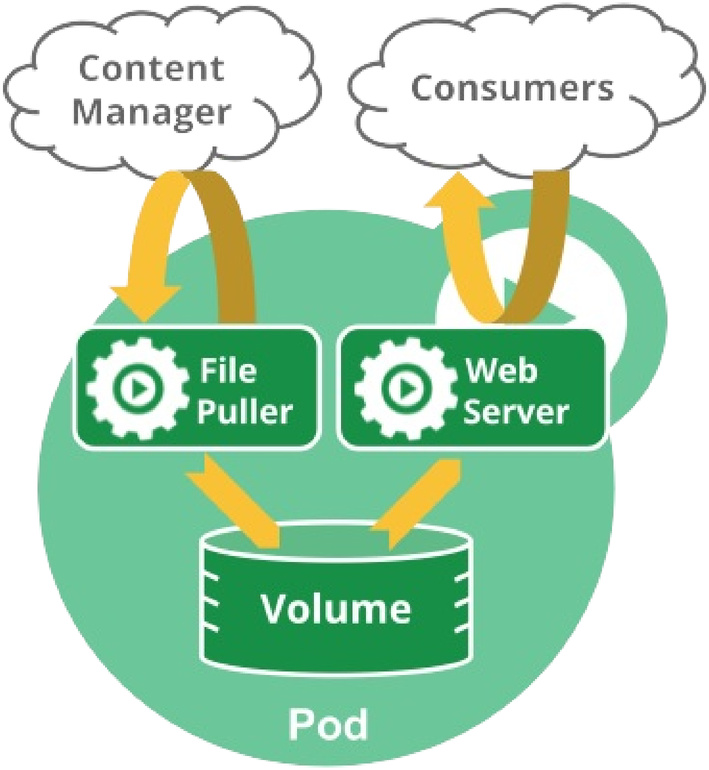
- The Distributed System Toolkit: Patterns for Composite Containers
- Container Design Patterns

每个Pod都是应用的一个实例。如果你想平行扩展应用的话（运行多个实例），你应该运行多个Pod，每个Pod都是一个应用实例。在Kubernetes中，这通常被称为replication。

Pod中如何管理多个容器

Pod中可以同时运行多个进程（作为容器运行）协同工作。同一个Pod中的容器会自动的分配到同一个 node 上。同一个Pod中的容器共享资源、网络环境和依赖，它们总是被同时调度。

注意在一个Pod中同时运行多个容器是一种比较高级的用法。只有当你的容器需要紧密配合协作的时候才考虑用这种模式。例如，你有一个容器作为web服务器运行，需要用到共享的volume，有另一个“sidecar”容器来从远端获取资源更新这些文件，如下图所示：



图片 - *pod diagram*

Pod中可以共享两种资源：网络和存储。

网络

每个Pod都会被分配一个唯一的IP地址。Pod中的所有容器共享网络空间，包括IP地址和端口。Pod内部的容器可以使用 `localhost` 互相通信。Pod中的容器与外界通信时，必须分配共享网络资源（例如使用宿主机的端口映射）。

存储

可以Pod指定多个共享的Volume。Pod中的所有容器都可以访问共享的volume。Volume也可以用来持久化Pod中的存储资源，以防容器重启后文件丢失。

使用Pod

你很少会直接在kubernetes中创建单个Pod。因为Pod的生命周期是短暂的，用后即焚的实体。当Pod被创建后（不论是由你直接创建还是被其他Controller），都会被Kuberentes调度到集群的Node上。直到Pod的进程终止、被删掉、因为缺少资源而被驱逐、或者Node故障之前这个Pod都会一直保持在那个Node上。

> 注意：重启Pod中的容器跟重启Pod不是一回事。Pod只提供容器的运行环境并保持容器的运行状态，重启容器不会造成Pod重启。

Pod不会自愈。如果Pod运行的Node故障，或者是调度器本身故障，这个Pod就会被删除。同样的，如果Pod所在Node缺少资源或者Pod处于维护状态，Pod也会被驱逐。Kubernetes使用更高级的称为Controller的抽象层，来管理Pod实例。虽然可以直接使用Pod，但是在Kubernetes中通常是使用Controller来管理Pod的。

Pod和Controller

Controller可以创建和管理多个Pod，提供副本管理、滚动升级和集群级别的自愈能力。例如，如果一个Node故障，Controller就能自动将该节点上的Pod调度到其他健康的Node上。

包含一个或者多个Pod的Controller示例：

- Deployment
- StatefulSet
- DaemonSet

通常，Controller会用你提供的Pod Template来创建相应的Pod。

Pod Templates

Pod模版是包含了其他object的Pod定义，例如Replication Controllers，Jobs和DaemonSets。Controller根据Pod模板来创建实际的Pod。

GitbookUpdated at 2018-05-09 14:15:47

Pod解析

Pod是kubernetes中可以创建的最小部署单元。

V1 core版本的Pod的配置模板见Pod template。

什么是Pod？

Pod就像是豌豆荚一样，它由一个或者多个容器组成（例如Docker容器），它们共享容器存储、网络和容器运行配置项。Pod中的容器总是被同时调度，有共同的运行环境。你可以把单个Pod想象成是运行独立应用的“逻辑主机”——其中运行着一个或者多个紧密耦合的应用容器——在有容器之前，这些应用都是运行在几个相同的物理机或者虚拟机上。

尽管kubernetes支持多种容器运行时，但是Docker依然是最常用的运行时环境，我们可以使用Docker的术语和规则来定义Pod。

Pod中共享的环境包括Linux的namespace，cgroup和其他可能的隔绝环境，这一点跟Docker容器一致。在Pod的环境中，每个容器中可能还有更小的子隔离环境。

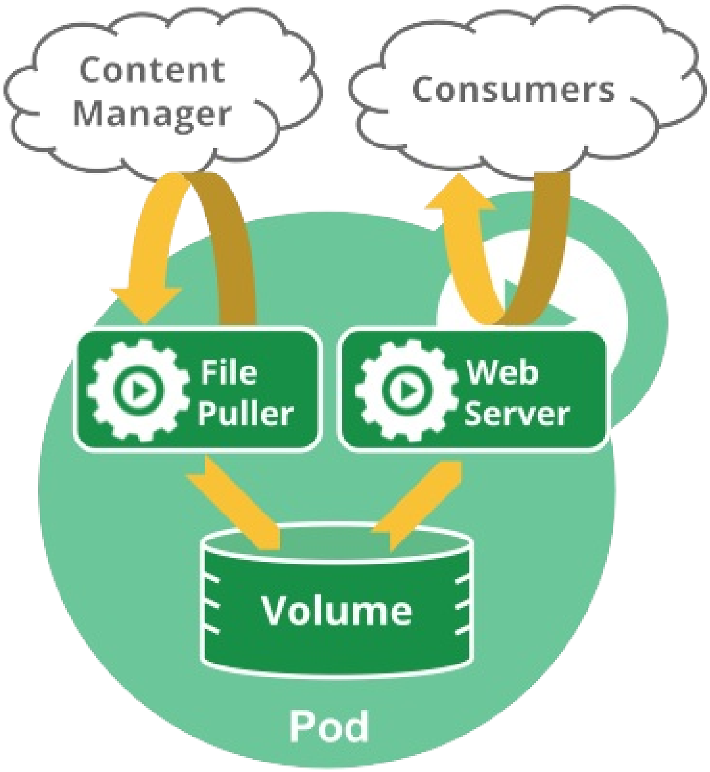
Pod中的容器共享IP地址和端口号，它们之间可以通过 `localhost` 互相发现。它们之间可以通过进程间通信，例如SystemV信号或者POSIX共享内存。不同Pod之间的容器具有不同的IP地址，不能直接通过IPC通信。

Pod中的容器也有访问共享volume的权限，这些volume会被定义成pod的一部分并挂载到应用容器的文件系统中。

根据Docker的结构，Pod中的容器共享namespace和volume，不支持共享PID的namespace。

就像每个应用容器，pod被认为是临时（非持久的）实体。在Pod的生命周期中讨论过，pod被创建后，被分配一个唯一的ID（UID），调度到节点上，并一致维持期望的状态直到被终结（根据重启策略）或者被删除。如果node死掉了，分配到了这个node上的pod，在经过一个超时时间后会被重新调度到其他node节点上。一个给定的pod（如UID定义的）不会被“重新调度”到新的节点上，而是被一个同样的pod取代，如果期望的话甚至可以是相同的名字，但是会有一个新的UID（查看replication controller获取详情）。（未来，一个更高级别的API将支持pod迁移）。

Volume跟pod有相同的生命周期（当其UID存在的时候）。当Pod因为某种原因被删除或者被新创建的相同的Pod取代，它相关的东西（例如volume）也会被销毁和再创建一个新的volume。



图片 - *Pod*示意图

A multi-container pod that contains a file puller and a web server that uses a persistent volume for shared storage between the containers.

Pod的动机

管理

Pod是一个服务的多个进程的聚合单位，pod提供这种模型能够简化应用部署管理，通过提供一个更高级别的抽象的方式。Pod作为一个独立的部署单位，支持横向扩展和复制。共生（协同调度），命运共同体（例如被终结），协同复制，资源共享，依赖管理，Pod都会自动的为容器处理这些问题。

资源共享和通信

Pod中的应用可以共享网络空间（IP地址和端口），因此可以通过 `localhost` 互相发现。因此，pod中的应用必须协调端口占用。每个pod都有一个唯一的IP地址，跟物理机和其他pod都处于一个扁平的网络空间中，它们之间可以直接连通。

Pod中应用容器的hostname被设置成Pod的名字。

Pod中的应用容器可以共享volume。Volume能够保证pod重启时使用的数据不丢失。

Pod的使用

Pod也可以用于垂直应用栈（例如LAMP），这样使用的主要动机是为了支持共同调度和协调管理应用程序，例如：

- content management systems, file and data loaders, local cache managers, etc.
- log and checkpoint backup, compression, rotation, snapshotting, etc.
- data change watchers, log tailers, logging and monitoring adapters, event publishers, etc.
- proxies, bridges, and adapters
- controllers, managers, configurators, and updaters

通常单个pod中不会同时运行一个应用的多个实例。

详细说明请看： The Distributed System ToolKit: Patterns for Composite Containers.

其他替代选择

为什么不直接在一个容器中运行多个应用程序呢？

1. 透明。让Pod中的容器对基础设施可见，以便基础设施能够为这些容器提供服务，例如进程管理和资源监控。这可以为用户带来极大的便利。
2. 解耦软件依赖。每个容器都可以进行版本管理，独立的编译和发布。未来kubernetes甚至可能支持单个容器的在线升级。
3. 使用方便。用户不必运行自己的进程管理器，还要担心错误信号传播等。
4. 效率。因为由基础架构提供更多的职责，所以容器可以变得更加轻量级。

为什么不支持容器的亲和性的协同调度？

这种方法可以提供容器的协同定位，能够根据容器的亲和性进行调度，但是无法实现使用pod带来的大部分好处，例如资源共享，IPC，保持状态一致性和简化管理等。

Pod的持久性（或者说缺乏持久性）

Pod在设计支持就不是作为持久化实体的。在调度失败、节点故障、缺少资源或者节点维护的状态下都会死掉会被驱逐。

通常，用户不需要手动直接创建Pod，而是应该使用controller（例如Deployments），即使是在创建单个Pod的情况下。Controller可以提供集群级别的自愈功能、复制和升级管理。

The use of collective APIs as the primary user-facing primitive is relatively common among cluster scheduling systems, including Borg, Marathon, Aurora, and Tupperware.

Pod is exposed as a primitive in order to facilitate:

- scheduler and controller pluggability
- support for pod-level operations without the need to "proxy" them via controller APIs
- decoupling of pod lifetime from controller lifetime, such as for bootstrapping
- decoupling of controllers and services — the endpoint controller just watches pods
- clean composition of Kubelet-level functionality with cluster-level functionality — Kubelet is effectively the "pod controller"
- high-availability applications, which will expect pods to be replaced in advance of their termination and certainly in advance of deletion, such as in the case of planned evictions, image prefetching, or live pod migration #3949

StatefulSet controller（目前还是beta状态）支持有状态的Pod。在1.4版本中被称为PetSet。在kubernetes之前的版本中创建有状态pod的最佳方式是创建一个replica为1的replication controller。

Pod的终止

因为Pod作为在集群的节点上运行的进程，所以在不再需要的时候能够优雅的终止掉是十分必要的（比起使用发送KILL信号这种暴力的方式）。用户需要能够放松删除请求，并且知道它们何时会被终止，是否被正确的删除。用户想终止程序时发送删除pod的请求，在pod可以被强制删除前会有一个优雅删除的时间，会发送一个TERM请求到每个容器的主进程。一旦超时，将向主进程发送KILL信号并从API server中删除。如果kubelet或者container manager在等待进程终止的过程中重启，在重启后仍然会重试完整的优雅删除阶段。

示例流程如下：

1. 用户发送删除pod的命令，默认优雅删除时期是30秒；
2. 在Pod超过该优雅删除期限后API server就会更新Pod的状态为“dead”；
3. 在客户端命令行上显示的Pod状态为“terminating”；
4. 跟第三步同时，当kubelet发现pod被标记为“terminating”状态时，开始停止pod进程：
    i. 如果在pod中定义了preStop hook，在停止pod前会被调用。如果在优雅删除期限过期后，preStop hook依然在运行，第二步会再增加2秒的优雅时间；
    ii. 向Pod中的进程发送TERM信号；
5. 跟第三步同时，该Pod将从该service的端点列表中删除，不再是replication controller的一部分。关闭的慢的pod将继续处理load balancer转发的流量；
6. 过了优雅周期后，将向Pod中依然运行的进程发送SIGKILL信号而杀掉进程。
7. Kublete会在API server中完成Pod的的删除，通过将优雅周期设置为0（立即删除）。Pod在API中消失，并且在客户端也不可见。

删除优雅周期默认是30秒。 `kubectl delete` 命令支持 `–grace-period=<seconds>` 选项，允许用户设置自己的优雅周期时间。如果设置为0将强制删除pod。在kubectl>=1.5版本的命令中，你必须同时使用 `--force` 和 `--grace-period=0` 来强制删除pod。

强制删除Pod

Pod的强制删除是通过在集群和etcd中将其定义为删除状态。当执行强制删除命令时，API server不会等待该pod所运行在节点上的kubelet确认，就会立即将该pod从API server中移除，这时就可以创建跟原pod同名的pod了。这时，在节点上的pod会被立即设置为terminating状态，不过在被强制删除之前依然有一小段优雅删除周期。

强制删除对于某些pod具有潜在危险性，请谨慎使用。使用StatefulSet pod的情况下，请参考删除StatefulSet中的pod文章。

Pod中容器的特权模式

从kubernetes1.1版本开始，pod中的容器就可以开启previleged模式，在容器定义文件的 `SecurityContext` 下使用 `privileged` flag。 这在使用Linux的网络操作和访问设备的能力时是很用的。同时开启的特权模式的Pod中的容器也可以访问到容器外的进程和应用。在不需要修改和重新编译kubelet的情况下就可以使用pod来开发节点的网络和存储插件。

如果master节点运行的是kuberentes1.1或更高版本，而node节点的版本低于1.1版本，则API server将也可以接受新的特权模式的pod，但是无法启动，pod将处于pending状态。

执行 `kubectl describe pod FooPodName` ，可以看到为什么pod处于pending状态。输出的event列表中将显示： `Error validating pod "FooPodName"."FooPodNamespace" from api, ignoring: spec.containers[0].securityContext.privileged: forbidden '<*>(0xc2089d3248)true'`

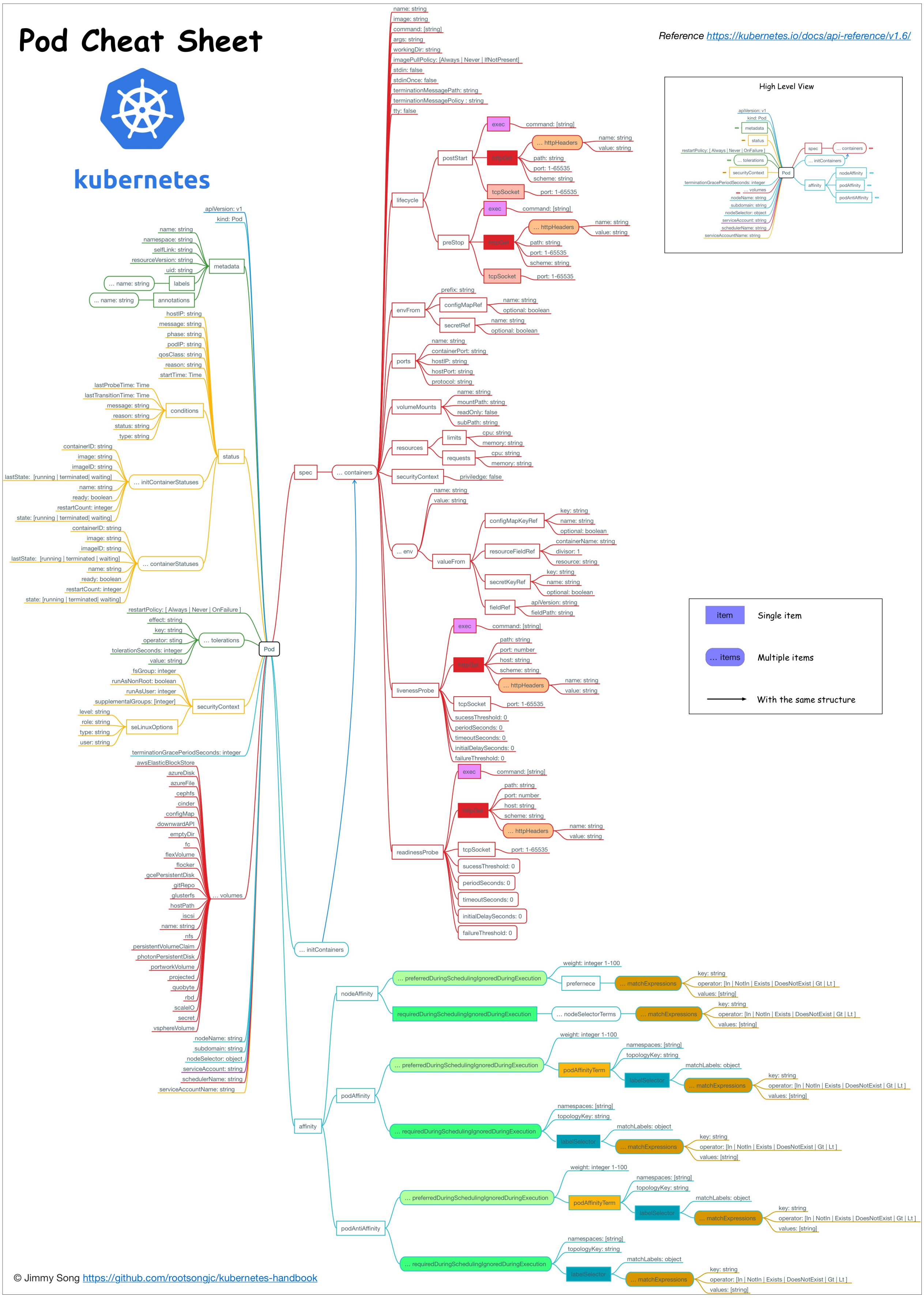
如果master节点的版本低于1.1，无法创建特权模式的pod。如果你仍然试图去创建的话，你得到如下错误：

`The Pod "FooPodName" is invalid. spec.containers[0].securityContext.privileged: forbidden '<*>(0xc20b222db0)true'`

API Object

Pod是kubernetes REST API中的顶级资源类型。

在kuberentes1.6的V1 core API版本中的Pod的数据结构如下图所示：



图片 - *Pod Cheatsheet*

Init 容器

该特性在 1.6 版本已经推出 beta 版本。Init 容器可以在 PodSpec 中同应用程序的 `containers` 数组一起来指定。 beta 注解的值将仍需保留，并覆盖 PodSpec 字段值。

本文讲解 Init 容器的基本概念，它是一种专用的容器，在应用程序容器启动之前运行，并包括一些应用镜像中不存在的实用工具和安装脚本。

理解 Init 容器

Pod 能够具有多个容器，应用运行在容器里面，但是它也可能有一个或多个先于应用容器启动的 Init 容器。

Init 容器与普通的容器非常像，除了如下两点：

- Init 容器总是运行到成功完成为止。
- 每个 Init 容器都必须在下一个 Init 容器启动之前成功完成。

如果 Pod 的 Init 容器失败，Kubernetes 会不断地重启该 Pod，直到 Init 容器成功为止。然而，如果 Pod 对应的 `restartPolicy` 为 Never，它不会重新启动。

指定容器为 Init 容器，在 PodSpec 中添加 `initContainers` 字段，以 v1.Container 类型对象的 JSON 数组的形式，还有 app 的 `containers` 数组。 Init 容器的状态在 `status.initContainerStatuses` 字段中以容器状态数组的格式返回（类似 `status.containerStatuses` 字段）。

与普通容器的不同之处

Init 容器支持应用容器的全部字段和特性，包括资源限制、数据卷和安全设置。 然而，Init 容器对资源请求和限制的处理稍有不同，在下面 资源 处有说明。 而且 Init 容器不支持 Readiness Probe，因为它们必须在 Pod 就绪之前运行完成。

如果为一个 Pod 指定了多个 Init 容器，那些容器会按顺序一次运行一个。 每个 Init 容器必须运行成功，下一个才能够运行。 当所有的 Init 容器运行完成时，Kubernetes 初始化 Pod 并像平常一样运行应用容器。

Init 容器能做什么？

因为 Init 容器具有与应用程序容器分离的单独镜像，所以它们的启动相关代码具有如下优势：

- 它们可以包含并运行实用工具，但是出于安全考虑，是不建议在应用程序容器镜像中包含这些实用工具的。
- 它们可以包含使用工具和定制化代码来安装，但是不能出现在应用程序镜像中。例如，创建镜像没必要 `FROM` 另一个镜像，只需要在安装过程中使用类似 `sed` 、 `awk` 、 `python` 或 `dig` 这样的工具。
- 应用程序镜像可以分离出创建和部署的角色，而没有必要联合它们构建一个单独的镜像。
- Init 容器使用 Linux Namespace，所以相对应用程序容器来说具有不同的文件系统视图。因此，它们能够具有访问 Secret 的权限，而应用程序容器则不能。
- 它们必须在应用程序容器启动之前运行完成，而应用程序容器是并行运行的，所以 Init 容器能够提供了一种简单的阻塞或延迟应用容器的启动的方法，直到满足了一组先决条件。

示例

下面是一些如何使用 Init 容器的想法：

- 等待一个 Service 创建完成，通过类似如下 shell 命令：

```
for i in {1..100}; do sleep 1; if dig myservice; then exit 0; fi; exit 1
```

- 将 Pod 注册到远程服务器，通过在命令中调用 API，类似如下：

```
curl -X POST http://$MANAGEMENT_SERVICE_HOST:$MANAGEMENT_SERVICE_PORT/register -d 'instance=$(<POD_NAME>)&ip=$(<POD_IP>)'
```

- 在启动应用容器之前等一段时间，使用类似 `sleep 60` 的命令。
- 克隆 Git 仓库到数据卷。

- 将配置值放到配置文件中，运行模板工具为主应用容器动态地生成配置文件。例如，在配置文件中存放 POD_IP 值，并使用 Jinja 生成主应用配置文件。

更多详细用法示例，可以在 StatefulSet 文档 和 生产环境 Pod 指南 中找到。

使用 Init 容器

下面是 Kubernetes 1.5 版本 yaml 文件，展示了一个具有 2 个 Init 容器的简单 Pod。 第一个等待 `myservice` 启动，第二个等待 `mydb` 启动。 一旦这两个 Service 都启动完成，Pod 将开始启动。

```
apiVersion: v1
kind: Pod
metadata:
  name: myapp-pod
  labels:
    app: myapp
  annotations:
    pod.beta.kubernetes.io/init-containers: '[
        {
            "name": "init-myservice",
            "image": "busybox",
            "command": ["sh", "-c", "until nslookup myservice; d
o echo waiting for myservice; sleep 2; done;"]
        },
        {
            "name": "init-mydb",
            "image": "busybox",
            "command": ["sh", "-c", "until nslookup mydb; do ech
o waiting for mydb; sleep 2; done;"]
        }
    ]'
spec:
  containers:
  - name: myapp-container
    image: busybox
    command: ['sh', '-c', 'echo The app is running! && sleep 360
0']
```

这是 Kubernetes 1.6 版本的新语法，尽管老的 annotation 语法仍然可以使用。我们已经把 Init 容器的声明移到 `spec` 中：

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: myapp-pod
  labels:
    app: myapp
spec:
  containers:
  - name: myapp-container
    image: busybox
    command: ['sh', '-c', 'echo The app is running! && sleep 360
0']
  initContainers:
  - name: init-myservice
    image: busybox
    command: ['sh', '-c', 'until nslookup myservice; do echo wai
ting for myservice; sleep 2; done;']
  - name: init-mydb
    image: busybox
    command: ['sh', '-c', 'until nslookup mydb; do echo waiting
for mydb; sleep 2; done;']
```

1.5 版本的语法在 1.6 版本仍然可以使用，但是我们推荐使用 1.6 版本的新语法。在 Kubernetes 1.6 版本中，Init 容器在 API 中新建了一个字段。 虽然期望使用 beta 版本的 annotation，但在未来发行版将会被废弃掉。

下面的 yaml 文件展示了 `mydb` 和 `myservice` 两个 Service：

```yaml
kind: Service
apiVersion: v1
metadata:
  name: myservice
spec:
  ports:
    - protocol: TCP
      port: 80
      targetPort: 9376
---
kind: Service
apiVersion: v1
metadata:
  name: mydb
spec:
  ports:
    - protocol: TCP
      port: 80
      targetPort: 9377
```

这个 Pod 可以使用下面的命令进行启动和调试：

```
$ kubectl create -f myapp.yaml
pod "myapp-pod" created
$ kubectl get -f myapp.yaml
NAME        READY     STATUS     RESTARTS   AGE
myapp-pod   0/1       Init:0/2   0          6m
$ kubectl describe -f myapp.yaml
Name:          myapp-pod
Namespace:     default
[...]
Labels:        app=myapp
Status:        Pending
[...]
Init Containers:
  init-myservice:
[...]
    State:         Running
[...]
  init-mydb:
[...]
    State:         Waiting
      Reason:      PodInitializing
    Ready:         False
[...]
Containers:
  myapp-container:
[...]
    State:         Waiting
      Reason:      PodInitializing
    Ready:         False
[...]
Events:
  FirstSeen    LastSeen    Count    From                      Su
bObjectPath                           Type          Reason
  Message
  ---------    --------    -----    ----                      --
-----------                           --------      ------
  -------
  16s          16s         1        {default-scheduler }
                                      Normal        Scheduled
  Successfully assigned myapp-pod to 172.17.4.201
  16s          16s         1        {kubelet 172.17.4.201}    sp
ec.initContainers{init-myservice}     Normal        Pulling
  pulling image "busybox"
  13s          13s         1        {kubelet 172.17.4.201}    sp
ec.initContainers{init-myservice}     Normal        Pulled
  Successfully pulled image "busybox"
  13s          13s         1        {kubelet 172.17.4.201}    sp
ec.initContainers{init-myservice}     Normal        Created
  Created container with docker id 5ced34a04634; Security:[secco
mp=unconfined]
```

```
  13s          13s          1         {kubelet 172.17.4.201}    sp
ec.initContainers{init-myservice}     Normal        Started
  Started container with docker id 5ced34a04634
$ kubectl logs myapp-pod -c init-myservice # Inspect the first i
nit container
$ kubectl logs myapp-pod -c init-mydb      # Inspect the second
init container
```

一旦我们启动了 `mydb` 和 `myservice` 这两个 Service，我们能够看到 Init 容器完成，并且 `myapp-pod` 被创建：

```
$ kubectl create -f services.yaml
service "myservice" created
service "mydb" created
$ kubectl get -f myapp.yaml
NAME        READY     STATUS    RESTARTS   AGE
myapp-pod   1/1       Running   0          9m
```

这个例子非常简单，但是应该能够为我们创建自己的 Init 容器提供一些启发。

具体行为

在 Pod 启动过程中，Init 容器会按顺序在网络和数据卷初始化之后启动。 每个容器必须在下一个容器启动之前成功退出。 如果由于运行时或失败退出，导致容器启动失败，它会根据 Pod 的 `restartPolicy` 指定的策略进行重试。 然而，如果 Pod 的 `restartPolicy` 设置为 Always，Init 容器失败时会使用 `RestartPolicy` 策略。

在所有的 Init 容器没有成功之前，Pod 将不会变成 `Ready` 状态。 Init 容器的端口将不会在 Service 中进行聚集。 正在初始化中的 Pod 处于 `Pending` 状态，但应该会将条件 `Initializing` 设置为 true。

如果 Pod 重启，所有 Init 容器必须重新执行。

对 Init 容器 spec 的修改，被限制在容器 image 字段中。 更改 Init 容器的 image 字段，等价于重启该 Pod。

因为 Init 容器可能会被重启、重试或者重新执行，所以 Init 容器的代码应该是幂等的。 特别地，被写到 `EmptyDirs` 中文件的代码，应该对输出文件可能已经存在做好准备。

Init 容器具有应用容器的所有字段。 除了 `readinessProbe` ，因为 Init 容器无法定义不同于完成（completion）的就绪（readiness）的之外的其他状态。 这会在验证过程中强制执行。

在 Pod 上使用 `activeDeadlineSeconds` ，在容器上使用 `livenessProbe` ，这样能够避免 Init 容器一直失败。 这就为 Init 容器活跃设置了一个期限。

在 Pod 中的每个 app 和 Init 容器的名称必须唯一；与任何其它容器共享同一个名称，会在验证时抛出错误。

资源

为 Init 容器指定顺序和执行逻辑，下面对资源使用的规则将被应用：

- 在所有 Init 容器上定义的，任何特殊资源请求或限制的最大值，是 有效初始请求/限制
- Pod 对资源的有效请求/限制要高于：
    - 所有应用容器对某个资源的请求/限制之和
    - 对某个资源的有效初始请求/限制
- 基于有效请求/限制完成调度，这意味着 Init 容器能够为初始化预留资源，这些资源在 Pod 生命周期过程中并没有被使用。

- Pod 的 有效 *QoS* 层，是 Init 容器和应用容器相同的 QoS 层。

基于有效 Pod 请求和限制来应用配额和限制。 Pod 级别的 cgroups 是基于有效 Pod 请求和限制，和调度器相同。

Pod 重启的原因

Pod 能够重启，会导致 Init 容器重新执行，主要有如下几个原因：

- 用户更新 PodSpec 导致 Init 容器镜像发生改变。应用容器镜像的变更只会重启应用容器。
- Pod 基础设施容器被重启。这不多见，但某些具有 root 权限可访问 Node 的人可能会这样做。
- 当 `restartPolicy` 设置为 Always，Pod 中所有容器会终止，强制重启，由于垃圾收集导致 Init 容器完整的记录丢失。

支持与兼容性

Apiserver 版本为 1.6 或更高版本的集群，通过使用 `spec.initContainers` 字段来支持 Init 容器。 之前的版本可以使用 alpha 和 beta 注解支持 Init 容器。 `spec.initContainers` 字段也被加入到 alpha 和 beta 注解中，所以 Kubernetes 1.3.0 版本或更高版本可以执行 Init 容器，并且 1.6 版本的 apiserver 能够安全地回退到 1.5.x 版本，而不会使存在的已创建 Pod 失去 Init 容器的功能。

原文地址：https://k8smeetup.github.io/docs/concepts/workloads/pods/init-containers/

译者：shirdrn

Pause容器

Pause容器，又叫Infra容器，本文将探究该容器的作用与原理。

我们知道在kubelet的配置中有这样一个参数：

```
KUBELET_POD_INFRA_CONTAINER=--pod-infra-container-image=registry
.access.redhat.com/rhel7/pod-infrastructure:latest
```

上面是openshift中的配置参数，kubernetes中默认的配置参数是：

```
KUBELET_POD_INFRA_CONTAINER=--pod-infra-container-image=gcr.io/g
oogle_containers/pause-amd64:3.0
```

Pause容器，是可以自己来定义，官方使用的 `gcr.io/google_containers/pause-amd64:3.0` 容器的代码见Github，使用C语言编写。

Pause容器的作用

我们检查nod节点的时候会发现每个node上都运行了很多的pause容器，例如如下。

```
$ docker ps
CONTAINER ID        IMAGE

             COMMAND                  CREATED             STATUS
              PORTS               NAMES
2c7d50f1a7be        docker.io/jimmysong/heapster-grafana-amd64@s
ha256:d663759b3de86cf62e64a43b021f133c383e8f7b0dc2bdd78115bc95db
371c9a       "/run.sh"                3 hours ago         Up 3 h
ours                              k8s_grafana_monitoring-influxd
b-grafana-v4-5697c6b59-76zqs_kube-system_5788a3c5-29c0-11e8-9e88
-525400005732_0
5df93dea877a        docker.io/jimmysong/heapster-influxdb-amd64@
sha256:a217008b68cb49e8f038c4eeb6029261f02adca81d8eae8c5c01d0303
61274b8      "influxd --config ..."   3 hours ago         Up 3 h
ours                              k8s_influxdb_monitoring-influx
db-grafana-v4-5697c6b59-76zqs_kube-system_5788a3c5-29c0-11e8-9e8
8-525400005732_0
9cec6c0ef583        jimmysong/pause-amd64:3.0

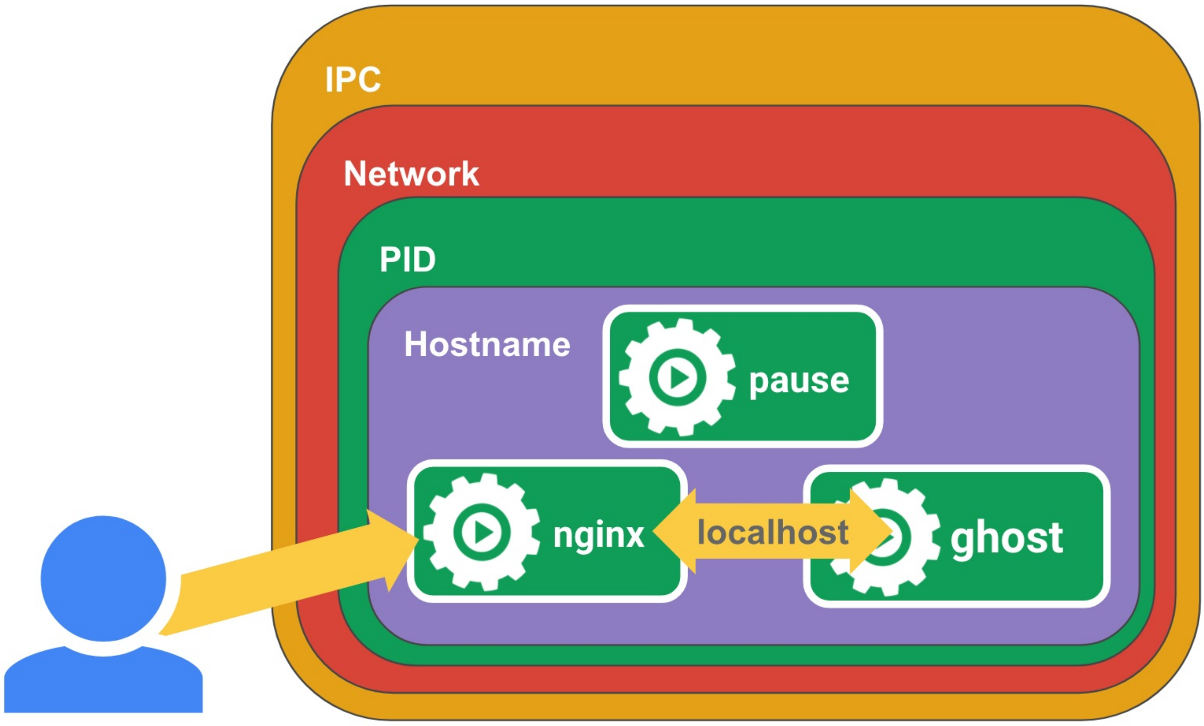
             "/pause"                 3 hours ago         Up 3 h
ours                              k8s_POD_monitoring-influxdb-gr
afana-v4-5697c6b59-76zqs_kube-system_5788a3c5-29c0-11e8-9e88-525
400005732_0
54d06e30a4c7        docker.io/jimmysong/kubernetes-dashboard-amd
64@sha256:668710d034c4209f8fa9a342db6d8be72b6cb5f1f3f696cee2379b
8512330be4   "/dashboard --inse..."   3 hours ago         Up 3 h
ours                              k8s_kubernetes-dashboard_kuber
netes-dashboard-65486f5fdf-lshl7_kube-system_27c414a1-29c0-11e8-
9e88-525400005732_0
5a5ef33b0d58        jimmysong/pause-amd64:3.0

             "/pause"                 3 hours ago         Up 3 h
ours                              k8s_POD_kubernetes-dashboard-6
5486f5fdf-lshl7_kube-system_27c414a1-29c0-11e8-9e88-525400005732
_0
```

kubernetes中的pause容器主要为每个业务容器提供以下功能：

- 在pod中担任Linux命名空间共享的基础；
- 启用pid命名空间，开启init进程。

在The Almighty Pause Container这篇文章中做出了详细的说明，pause容器的作用可以从这个例子中看出，首先见下图：



图片 - *Pause*容器

我们首先在节点上运行一个pause容器。

```
docker run -d --name pause -p 8880:80 jimmysong/pause-amd64:3.0
```

然后再运行一个nginx容器，nginx将为 `localhost:2368` 创建一个代理。

```
$ cat <<EOF >> nginx.conff
error_log stderr;
events { worker_connections  1024; }
http {
    access_log /dev/stdout combined;
    server {
        listen 80 default_server;
        server_name example.com www.example.com;
        location / {
            proxy_pass http://127.0.0.1:2368;
        }
    }
}
EOF
$ docker run -d --name nginx -v `pwd`/nginx.conf:/etc/nginx/ngin
x.conf --net=container:pause --ipc=container:pause --pid=contain
er:pause nginx
```

然后再为ghost创建一个应用容器，这是一款博客软件。

```
$ docker run -d --name ghost --net=container:pause --ipc=contain
er:pause --pid=container:pause ghost
```

现在访问http://localhost:8880/就可以看到ghost博客的界面了。

解析

pause容器将内部的80端口映射到宿主机的8880端口，pause容器在宿主机上设置好了网络namespace后，nginx容器加入到该网络namespace中，我们看到nginx容器启动的时候指定了 `--net=container:pause` ，ghost容器同样加入到了该网络namespace中，这样三个容器就共享了网络，互相之间就可以使用 `localhost` 直接通信， `--ipc=contianer:pause --pid=container:pause` 就是三个容器处于同一个namespace中，init进程为 `pause` ，这时我们进入到ghost容器中查看进程情况。

```
# ps aux
USER        PID %CPU %MEM     VSZ   RSS TTY      STAT START   TIME
 COMMAND
root          1  0.0  0.0    1024     4 ?        Ss   13:49   0:00
 /pause
root          5  0.0  0.1   32432  5736 ?        Ss   13:51   0:00
 nginx: master p
systemd+      9  0.0  0.0   32980  3304 ?        S    13:51   0:00
 nginx: worker p
node         10  0.3  2.0 1254200 83788 ?        Ssl  13:53   0:03
 node current/in
root         79  0.1  0.0    4336   812 pts/0    Ss   14:09   0:00
 sh
root         87  0.0  0.0   17500  2080 pts/0    R+   14:10   0:00
ps aux
```

在ghost容器中同时可以看到pause和nginx容器的进程，并且pause容器的PID是1。而在kubernetes中容器的PID=1的进程即为容器本身的业务进程。

参考

- The Almighty Pause Container
- Kubernetes之Pause容器

Pod 安全策略

`PodSecurityPolicy` 类型的对象能够控制，是否可以向 Pod 发送请求，该 Pod 能够影响被应用到 Pod 和容器的 `SecurityContext` 。 查看 Pod 安全策略建议 获取更多信息。

什么是 Pod 安全策略？

Pod 安全策略 是集群级别的资源，它能够控制 Pod 运行的行为，以及它具有访问什么的能力。 `PodSecurityPolicy` 对象定义了一组条件，指示 Pod 必须按系统所能接受的顺序运行。 它们允许管理员控制如下方面：

控制面	字段名称
已授权容器的运行	`privileged`
为容器添加默认的一组能力	`defaultAddCapabilities`
为容器去掉某些能力	`requiredDropCapabilities`
容器能够请求添加某些能力	`allowedCapabilities`
控制卷类型的使用	`volumes`
主机网络的使用	`hostNetwork`
主机端口的使用	`hostPorts`
主机 PID namespace 的使用	`hostPID`
主机 IPC namespace 的使用	`hostIPC`
主机路径的使用	`allowedHostPaths`
容器的 SELinux 上下文	`seLinux`
用户 ID	`runAsUser`
配置允许的补充组	`supplementalGroups`
分配拥有 Pod 数据卷的 FSGroup	`fsGroup`
必须使用一个只读的 root 文件系统	`readOnlyRootFilesystem`

Pod 安全策略 由设置和策略组成，它们能够控制 Pod 访问的安全特征。这些设置分为如下三类：

- 基于布尔值控制：这种类型的字段默认为最严格限制的值。
- 基于被允许的值集合控制：这种类型的字段会与这组值进行对比，以确认值被允许。
- 基于策略控制：设置项通过一种策略提供的机制来生成该值，这种机制能够确保指定的值落在被允许的这组值中。

RunAsUser

- *MustRunAs* - 必须配置一个 `range` 。使用该范围内的第一个值作为默认值。验证是否不在配置的该范围内。
- *MustRunAsNonRoot* - 要求提交的 Pod 具有非零 `runAsUser` 值，或在镜像中定义了 `USER` 环境变量。不提供默认值。
- *RunAsAny* - 没有提供默认值。允许指定任何 `runAsUser` 。

SELinux

- *MustRunAs* - 如果没有使用预分配的值，必须配置 `seLinuxOptions` 。默认使用 `seLinuxOptions` 。验证 `seLinuxOptions` 。
- *RunAsAny* - 没有提供默认值。允许任意指定的 `seLinuxOptions` ID。

SupplementalGroups

- *MustRunAs* - 至少需要指定一个范围。默认使用第一个范围的最小值。验证所有范围的值。
- *RunAsAny* - 没有提供默认值。允许任意指定的 `supplementalGroups` ID。

FSGroup

- *MustRunAs* - 至少需要指定一个范围。默认使用第一个范围的最小值。验证在第一个范围内的第一个 ID。
- *RunAsAny* - 没有提供默认值。允许任意指定的 `fsGroup` ID。

控制卷

通过设置 PSP 卷字段，能够控制具体卷类型的使用。当创建一个卷的时候，与该字段相关的已定义卷可以允许设置如下值：

1. azureFile
2. azureDisk
3. flocker
4. flexVolume
5. hostPath
6. emptyDir
7. gcePersistentDisk
8. awsElasticBlockStore
9. gitRepo
10. secret
11. nfs
12. iscsi
13. glusterfs
14. persistentVolumeClaim
15. rbd
16. cinder
17. cephFS
18. downwardAPI
19. fc
20. configMap
21. vsphereVolume
22. quobyte
23. photonPersistentDisk
24. projected
25. portworxVolume
26. scaleIO
27. storageos
28. * (allow all volumes)

对新的 PSP，推荐允许的卷的最小集合包括：configMap、downwardAPI、emptyDir、persistentVolumeClaim、secret 和 projected。

主机网络

- *HostPorts*， 默认为 `empty` 。 `HostPortRange` 列表通过 `min` (包含) and `max` (包含) 来定义，指定了被允许的主机端口。

允许的主机路径

- *AllowedHostPaths* 是一个被允许的主机路径前缀的白名单。空值表示所有的主机路径都可以使用。

许可

包含 `PodSecurityPolicy` 的 许可控制，允许控制集群资源的创建和修改，基于这些资源在集群范围内被许可的能力。

许可使用如下的方式为 Pod 创建最终的安全上下文：

1. 检索所有可用的 PSP。
2. 生成在请求中没有指定的安全上下文设置的字段值。
3. 基于可用的策略，验证最终的设置。

如果某个策略能够匹配上，该 Pod 就被接受。如果请求与 PSP 不匹配，则 Pod 被拒绝。

Pod 必须基于 PSP 验证每个字段。

创建 Pod 安全策略

下面是一个 Pod 安全策略的例子，所有字段的设置都被允许：

```yaml
apiVersion: extensions/v1beta1
kind: PodSecurityPolicy
metadata:
  name: permissive
spec:
  seLinux:
    rule: RunAsAny
  supplementalGroups:
    rule: RunAsAny
  runAsUser:
    rule: RunAsAny
  fsGroup:
    rule: RunAsAny
  hostPorts:
  - min: 8000
    max: 8080
  volumes:
  - '*'
```

下载示例文件可以创建该策略，然后执行如下命令：

```
$ kubectl create -f ./psp.yaml
podsecuritypolicy "permissive" created
```

获取 Pod 安全策略列表

获取已存在策略列表，使用 `kubectl get` ：

```
$ kubectl get psp
NAME        PRIV    CAPS   SELINUX    RUNASUSER          FSGROUP    S
UPGROUP  READONLYROOTFS  VOLUMES
permissive  false   []     RunAsAny   RunAsAny           RunAsAny   R
unAsAny  false           [*]
privileged  true    []     RunAsAny   RunAsAny           RunAsAny   R
unAsAny  false           [*]
restricted  false   []     RunAsAny   MustRunAsNonRoot   RunAsAny   R
unAsAny  false           [emptyDir secret downwardAPI configMap
persistentVolumeClaim projected]
```

修改 Pod 安全策略

通过交互方式修改策略，使用 `kubectl edit` ：

```
$ kubectl edit psp permissive
```

该命令将打开一个默认文本编辑器，在这里能够修改策略。

删除 Pod 安全策略

一旦不再需要一个策略，很容易通过 `kubectl` 删除它：

```
$ kubectl delete psp permissive
podsecuritypolicy "permissive" deleted
```

启用 Pod 安全策略

为了能够在集群中使用 Pod 安全策略，必须确保如下：

1. 启用 API 类型 `extensions/v1beta1/podsecuritypolicy` （仅对 1.6 之前的版本）
2. 启用许可控制器 `PodSecurityPolicy`
3. 定义自己的策略

使用 RBAC

在 Kubernetes 1.5 或更新版本，可以使用 PodSecurityPolicy 来控制，对基于用户角色和组的已授权容器的访问。访问不同的 PodSecurityPolicy 对象，可以基于认证来控制。基于 Deployment、ReplicaSet 等创建的 Pod，限制访问 PodSecurityPolicy 对象，Controller Manager 必须基于安全 API 端口运行，并且不能够具有超级用户权限。

PodSecurityPolicy 认证使用所有可用的策略，包括创建 Pod 的用户，Pod 上指定的服务账户（service acount）。当 Pod 基于 Deployment、ReplicaSet 创建时，它是创建 Pod 的 Controller Manager，所以如果基于非安全 API 端口运行，允许所有的 PodSecurityPolicy 对象，并且不能够有效地实现细分权限。用户访问给定的 PSP 策略有效，仅当是直接部署 Pod 的情况。更多详情，查看 PodSecurityPolicy RBAC 示例，当直接部署 Pod 时，应用 PodSecurityPolicy 控制基于角色和组的已授权容器的访问 。

原文地址：https://k8smeetup.github.io/docs/concepts/policy/pod-security-policy/

译者：shirdrn

Pod 的生命周期

Pod phase

Pod 的 `status` 在信息保存在 PodStatus 中定义，其中有一个 `phase` 字段。

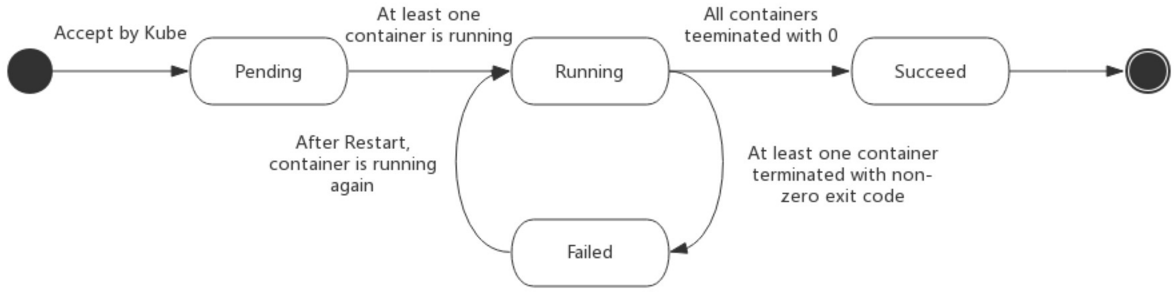
Pod 的相位（phase）是 Pod 在其生命周期中的简单宏观概述。该阶段并不是对容器或 Pod 的综合汇总，也不是为了做为综合状态机。

Pod 相位的数量和含义是严格指定的。除了本文档中列举的状态外，不应该再假定 Pod 有其他的 `phase` 值。

下面是 `phase` 可能的值：

- 挂起（Pending）：Pod 已被 Kubernetes 系统接受，但有一个或者多个容器镜像尚未创建。等待时间包括调度 Pod 的时间和通过网络下载镜像的时间，这可能需要花点时间。
- 运行中（Running）：该 Pod 已经绑定到了一个节点上，Pod 中所有的容器都已被创建。至少有一个容器正在运行，或者正处于启动或重启状态。
- 成功（Successed）：Pod 中的所有容器都被成功终止，并且不会再重启。
- 失败（Failed）：Pod 中的所有容器都已终止了，并且至少有一个容器是因为失败终止。也就是说，容器以非0状态退出或者被系统终止。
- 未知（Unkonwn）：因为某些原因无法取得 Pod 的状态，通常是因为与 Pod 所在主机通信失败。

下图是Pod的生命周期示意图，从图中可以看到Pod状态的变化。



图片 - *Pod*的生命周期示意图（图片来自网络）

Pod 状态

Pod 有一个 PodStatus 对象，其中包含一个 PodCondition 数组。 PodCondition 数组的每个元素都有一个 `type` 字段和一个 `status` 字段。`type` 字段是字符串，可能的值有 PodScheduled、Ready、Initialized 和 Unschedulable。`status` 字段是一个字符串，可能的值有 True、False 和 Unknown。

容器探针

探针 是由 kubelet 对容器执行的定期诊断。要执行诊断，kubelet 调用由容器实现的 Handler。有三种类型的处理程序：

- ExecAction：在容器内执行指定命令。如果命令退出时返回码为 0 则认为诊断成功。
- TCPSocketAction：对指定端口上的容器的 IP 地址进行 TCP 检查。如果端口打开，则诊断被认为是成功的。
- HTTPGetAction：对指定的端口和路径上的容器的 IP 地址执行 HTTP Get 请求。如果响应的状态码大于等于200 且小于 400，则诊断被认为是成功的。

每次探测都将获得以下三种结果之一：

- 成功：容器通过了诊断。
- 失败：容器未通过诊断。
- 未知：诊断失败，因此不会采取任何行动。

Kubelet 可以选择是否执行在容器上运行的两种探针执行和做出反应：

- `livenessProbe`：指示容器是否正在运行。如果存活探测失败，则 kubelet 会杀死容器，并且容器将受到其 重启策略 的影响。如果容器不提供存活探针，则默认状态为 `Success`。
- `readinessProbe`：指示容器是否准备好服务请求。如果就绪探测失败，端点控制器将从与 Pod 匹配的所有 Service 的端点中删除该 Pod 的 IP 地址。初始延迟之前的就绪状态默认为 `Failure`。如果容器不提供就绪探针，则默认状态为 `Success`。

该什么时候使用存活（liveness）和就绪（readiness）探针?

如果容器中的进程能够在遇到问题或不健康的情况下自行崩溃，则不一定需要存活探针; kubelet 将根据 Pod 的 `restartPolicy` 自动执行正确的操作。

如果您希望容器在探测失败时被杀死并重新启动，那么请指定一个存活探针，并指定 `restartPolicy` 为 Always 或 OnFailure。

如果要仅在探测成功时才开始向 Pod 发送流量，请指定就绪探针。在这种情况下，就绪探针可能与存活探针相同，但是 spec 中的就绪探针的存在意味着 Pod 将在没有接收到任何流量的情况下启动，并且只有在探针探测成功后才开始接收流量。

如果您希望容器能够自行维护，您可以指定一个就绪探针，该探针检查与存活探针不同的端点。

请注意，如果您只想在 Pod 被删除时能够排除请求，则不一定需要使用就绪探针；在删除 Pod 时，Pod 会自动将自身置于未完成状态，无论就绪探针是否存在。当等待 Pod 中的容器停止时，Pod 仍处于未完成状态。

Pod 和容器状态

有关 Pod 容器状态的详细信息，请参阅 PodStatus 和 ContainerStatus。请注意，报告的 Pod 状态信息取决于当前的 ContainerState。

重启策略

PodSpec 中有一个 `restartPolicy` 字段，可能的值为 Always、OnFailure 和 Never。默认为 Always。 `restartPolicy` 适用于 Pod 中的所有容器。 `restartPolicy` 仅指通过同一节点上的 kubelet 重新启动容器。失败的容器由 kubelet 以五分钟为上限的指数退避延迟（10秒，20秒，40秒...）重新启动，并在成功执行十分钟后重置。如 Pod 文档 中所述，一旦绑定到一个节点，Pod 将永远不会重新绑定到另一个节点。

Pod 的生命

一般来说，Pod 不会消失，直到人为销毁他们。这可能是一个人或控制器。这个规则的唯一例外是成功或失败的 `phase` 超过一段时间（由 master 确定）的Pod将过期并被自动销毁。

有三种可用的控制器：

- 使用 Job 运行预期会终止的 Pod，例如批量计算。Job 仅适用于重启策略为 `OnFailure` 或 `Never` 的 Pod。
- 对预期不会终止的 Pod 使用 ReplicationController、ReplicaSet 和 Deployment ，例如 Web 服务器。 ReplicationController 仅适用于具有 `restartPolicy` 为 Always 的 Pod。
- 提供特定于机器的系统服务，使用 DaemonSet 为每台机器运行一个 Pod 。

所有这三种类型的控制器都包含一个 PodTemplate。建议创建适当的控制器，让它们来创建 Pod，而不是直接自己创建 Pod。这是因为单独的 Pod 在机器故障的情况下没有办法自动复原，而控制器却可以。

如果节点死亡或与集群的其余部分断开连接，则 Kubernetes 将应用一个策略将丢失节点上的所有 Pod 的 `phase` 设置为 Failed。

示例

高级 liveness 探针示例

存活探针由 kubelet 来执行，因此所有的请求都在 kubelet 的网络命名空间中进行。

```yaml
apiVersion: v1
kind: Pod
metadata:
  labels:
    test: liveness
  name: liveness-http
spec:
  containers:
  - args:
    - /server
    image: gcr.io/google_containers/liveness
    livenessProbe:
      httpGet:
        # when "host" is not defined, "PodIP" will be used
        # host: my-host
        # when "scheme" is not defined, "HTTP" scheme will be us
ed. Only "HTTP" and "HTTPS" are allowed
        # scheme: HTTPS
        path: /healthz
        port: 8080
        httpHeaders:
          - name: X-Custom-Header
            value: Awesome
      initialDelaySeconds: 15
      timeoutSeconds: 1
    name: liveness
```

状态示例

- Pod 中只有一个容器并且正在运行。容器成功退出。
  - 记录完成事件。
  - 如果 `restartPolicy` 为：
    - Always：重启容器；Pod `phase` 仍为 Running。
    - OnFailure：Pod `phase` 变成 Succeeded。
    - Never：Pod `phase` 变成 Succeeded。
- Pod 中只有一个容器并且正在运行。容器退出失败。
  - 记录失败事件。
  - 如果 `restartPolicy` 为：
    - Always：重启容器；Pod `phase` 仍为 Running。
    - OnFailure：重启容器；Pod `phase` 仍为 Running。
    - Never：Pod `phase` 变成 Failed。
- Pod 中有两个容器并且正在运行。有一个容器退出失败。

  - 记录失败事件。
  - 如果 restartPolicy 为：
    - Always：重启容器；Pod `phase` 仍为 Running。
    - OnFailure：重启容器；Pod `phase` 仍为 Running。
    - Never：不重启容器；Pod `phase` 仍为 Running。
  - 如果有一个容器没有处于运行状态，并且两个容器退出：
    - 记录失败事件。
    - 如果 `restartPolicy` 为：
      - Always：重启容器；Pod `phase` 仍为 Running。
      - OnFailure：重启容器；Pod `phase` 仍为 Running。
      - Never：Pod `phase` 变成 Failed。
- Pod 中只有一个容器并处于运行状态。容器运行时内存超出限制：
  - 容器以失败状态终止。
  - 记录 OOM 事件。
  - 如果 `restartPolicy` 为：
    - Always：重启容器；Pod `phase` 仍为 Running。
    - OnFailure：重启容器；Pod `phase` 仍为 Running。
    - Never: 记录失败事件；Pod `phase` 仍为 Failed。
- Pod 正在运行，磁盘故障：
  - 杀掉所有容器。
  - 记录适当事件。
  - Pod `phase` 变成 Failed。
  - 如果使用控制器来运行，Pod 将在别处重建。
- Pod 正在运行，其节点被分段。
  - 节点控制器等待直到超时。
  - 节点控制器将 Pod `phase` 设置为 Failed。
  - 如果是用控制器来运行，Pod 将在别处重建。

原文地址：https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/

翻译：rootsongjc

Pod hook

Pod hook（钩子）是由Kubernetes管理的kubelet发起的，当容器中的进程启动前或者容器中的进程终止之前运行，这是包含在容器的生命周期之中。可以同时为Pod中的所有容器都配置hook。

Hook的类型包括两种：

- exec：执行一段命令
- HTTP：发送HTTP请求。

参考下面的配置：

```
apiVersion: v1
kind: Pod
metadata:
  name: lifecycle-demo
spec:
  containers:
  - name: lifecycle-demo-container
    image: nginx
    lifecycle:
      postStart:
        exec:
          command: ["/bin/sh", "-c", "echo Hello from the postSt
art handler > /usr/share/message"]
      preStop:
        exec:
          command: ["/usr/sbin/nginx","-s","quit"]
```

在容器创建之后，容器的Entrypoint执行之前，这时候Pod已经被调度到某台node上，被某个kubelet管理了，这时候kubelet会调用postStart操作，该操作跟容器的启动命令是在异步执行的，也就是说在postStart操作执行完成之前，kubelet会锁住容器，不让应用程序的进程启动，只有在 postStart操作完成之后容器的状态才会被设置成为RUNNING。

如果postStart或者preStop hook失败，将会终止容器。

调试hook

Hook调用的日志没有暴露个给Pod的event，所以只能通过 `describe` 命令来获取，如果有错误将可以看到 `FailedPostStartHook` 或 `FailedPreStopHook` 这样的event。

参考

- Attach Handlers to Container Lifecycle Events
- Container Lifecycle Hooks

Pod Preset

> 注意：PodPreset 资源对象只有 kubernetes 1.8 以上版本才支持。

Preset 就是预设，有时候想要让一批容器在启动的时候就注入一些信息，比如 secret、volume、volume mount 和环境变量，而又不想一个一个的改这些 Pod 的 tmeplate，这时候就可以用到 PodPreset 这个资源对象了。

本页是关于 PodPreset 的概述，该对象用来在 Pod 创建的时候向 Pod 中注入某些特定信息。该信息可以包括 secret、volume、volume mount 和环境变量。

理解 Pod Preset

`Pod Preset` 是用来在 Pod 被创建的时候向其中注入额外的运行时需求的 API 资源。

您可以使用 label selector 来指定为哪些 Pod 应用 Pod Preset。

使用 Pod Preset 使得 pod 模板的作者可以不必为每个 Pod 明确提供所有信息。这样一来，pod 模板的作者就不需要知道关于该服务的所有细节。

关于该背景的更多信息，请参阅 PodPreset 的设计方案。

如何工作

Kubernetes 提供了一个准入控制器（`PodPreset`），当其启用时，Pod Preset 会将应用创建请求传入到该控制器上。当有 Pod 创建请求发生时，系统将执行以下操作：

1. 检索所有可用的 `PodPresets`。
2. 检查 PodPreset 标签选择器上的标签，看看其是否能够匹配正在创建的 Pod 上的标签。
3. 尝试将由 `PodPreset` 定义的各种资源合并到正在创建的 Pod 中。
4. 出现错误时，在该 Pod 上引发记录合并错误的事件，PodPreset 不会注入任何资源到创建的 Pod 中。
5. 注释刚生成的修改过的 Pod spec，以表明它已被 PodPreset 修改过。注释的

格式为 `podpreset.admission.kubernetes.io/podpreset-<pod-preset name>": "<resource version>"` 。

每个 Pod 可以匹配零个或多个 Pod Prestet；并且每个 `PodPreset` 可以应用于零个或多个 Pod。 `PodPreset` 应用于一个或多个 Pod 时，Kubernetes 会修改 Pod Spec。对于 `Env` 、 `EnvFrom` 和 `VolumeMounts` 的更改，Kubernetes 修改 Pod 中所有容器的容器 spec；对于 `Volume` 的更改，Kubernetes 修改 Pod Spec。

> 注意：Pod Preset 可以在适当的时候修改 Pod spec 中的 `spec.containers` 字段。Pod Preset 中的资源定义将不会应用于 `initContainers` 字段。

禁用特定 Pod 的 Pod Preset

在某些情况下，您可能不希望 Pod 被任何 Pod Preset 所改变。在这些情况下，您可以在 Pod 的 Pod Spec 中添加注释： `podpreset.admission.kubernetes.io/exclude："true"` 。

启用 Pod Preset

为了在群集中使用 Pod Preset，您必须确保以下内容：

1. 您已启用 `settings.k8s.io/v1alpha1/podpreset` API 类型。例如，可以通过在 API server 的 `--runtime-config` 选项中包含 `settings.k8s.io/v1alpha1=true` 来完成此操作。
2. 您已启用 `PodPreset` 准入控制器。 一种方法是将 `PodPreset` 包含在为 API server 指定的 `--admission-control` 选项值中。
3. 您已经在要使用的命名空间中通过创建 `PodPreset` 对象来定义 `PodPreset` 。

更多资料

- 使用 PodPreset 向 Pod 中注入数据

本文为 Kubernetes 官方中文文档，地址：https://kubernetes.io/cn/docs/concepts/workloads/pods/podpreset/

翻译: rootsongjc

Pod中断与PDB（Pod中断预算）

这篇文档适用于要构建高可用应用程序的所有者，因此他们需要了解 Pod 可能发生什么类型的中断。也适用于要执行自动集群操作的集群管理员，如升级和集群自动扩容。

自愿中断和非自愿中断

Pod 不会消失，直到有人（人类或控制器）将其销毁，或者当出现不可避免的硬件或系统软件错误。

我们把这些不可避免的情况称为应用的非自愿性中断。例如：

- 后端节点物理机的硬件故障
- 集群管理员错误地删除虚拟机（实例）
- 云提供商或管理程序故障使虚拟机消失
- 内核恐慌（kernel panic）
- 节点由于集群网络分区而从集群中消失
- 由于节点资源不足而将容器逐出

除资源不足的情况外，大多数用户应该都熟悉以下这些情况；它们不是特定于 Kubernetes 的。

我们称这些情况为"自愿中断“。包括由应用程序所有者发起的操作和由集群管理员发起的操作。典型的应用程序所有者操作包括：

- 删除管理该 pod 的 Deployment 或其他控制器
- 更新了 Deployment 的 pod 模板导致 pod 重启
- 直接删除 pod（意外删除）

集群管理员操作包括：

- 排空（drain）节点进行修复或升级。
- 从集群中排空节点以缩小集群（了解集群自动调节）。
- 从节点中移除一个 pod，以允许其他 pod 使用该节点。

这些操作可能由集群管理员直接执行，也可能由集群管理员或集群托管提供商自动执行。

询问您的集群管理员或咨询您的云提供商或发行文档，以确定是否为您的集群启用了任何自动中断源。如果没有启用，您可以跳过创建 Pod Disruption Budget（Pod中断预算）。

处理中断

以下是一些减轻非自愿性中断的方法：

- 确保您的 pod 请求所需的资源。
- 如果您需要更高的可用性，请复制您的应用程序。（了解有关运行复制的无状态和有状态应用程序的信息。）
- 为了在运行复制应用程序时获得更高的可用性，请跨机架（使用反亲和性）或跨区域（如果使用多区域集群）分布应用程序。

自愿中断的频率各不相同。在 Kubernetes 集群上，根本没有自愿的中断。但是，您的集群管理员或托管提供商可能会运行一些导致自愿中断的附加服务。例如，节点软件更新可能导致自愿更新。另外，集群（节点）自动缩放的某些实现可能会导致碎片整理和紧缩节点的自愿中断。您的集群管理员或主机提供商应该已经记录了期望的自愿中断级别（如果有的话）。

Kubernetes 提供的功能可以满足在频繁地自动中断的同时运行高可用的应用程序。我们称之为“中断预算”。

中断预算的工作原理

应用程序所有者可以为每个应用程序创建一个 `PodDisruptionBudget` 对象（PDB）。PDB 将限制在同一时间自愿中断的复制应用程序中宕机的 Pod 的数量。例如，基于定额的应用程序希望确保运行的副本数量永远不会低于仲裁所需的数量。Web 前端可能希望确保提供负载的副本的数量永远不会低于总数的某个百分比。

集群管理器和托管提供商应使用遵循 `Pod Disruption Budgets` 的工具，方法是调用Eviction API而不是直接删除 Pod。例如 `kubectl drain` 命令和 Kubernetes-on-GCE 集群升级脚本（`cluster/gce/upgrade.sh`）。

当集群管理员想要排空节点时，可以使用 `kubectl drain` 命令。该命令会试图驱逐机器上的所有 pod。驱逐请求可能会暂时被拒绝，并且该工具会定期重试所有失败的请求，直到所有的 pod 都被终止，或者直到达到配置的超时时间。

PDB 指定应用程序可以容忍的副本的数量，相对于应该有多少副本。例如，具有 `spec.replicas：5` 的 Deployment 在任何给定的时间都应该有 5 个 Pod。如果其 PDB 允许在某一时刻有 4 个副本，那么驱逐 API 将只允许仅有一个而不是两个 Pod 自愿中断。

使用标签选择器来指定应用程序的一组 pod，这与应用程序的控制器（Deployment、StatefulSet 等）使用的相同。

Pod 控制器的 `.spec.replicas` 计算“预期的” pod 数量。使用对象的 `.metadata.ownerReferences` 值从控制器获取。

PDB 不能阻止非自愿中断的发生，但是它们确实会影响预算。

由于应用程序的滚动升级而被删除或不可用的 Pod 确实会计入中断预算，但控制器（如 Deployment 和 StatefulSet）在进行滚动升级时不受 PDB 的限制——在应用程序更新期间的故障处理是在控制器的规格（spec）中配置（了解更新 Deployment）。

使用驱逐 API 驱逐 pod 时，pod 会被优雅地终止（请参阅 PodSpec 中的 `terminationGracePeriodSeconds` ）。

PDB 示例

假设集群有3个节点， `node-1` 到 `node-3` 。集群中运行了一些应用，其中一个应用有3个副本，分别是 `pod-a` 、 `pod-b` 和 `pod-c` 。另外，还有一个与它相关的不具有 PDB 的 pod，我们称为之为 `pod-x` 。最初，所有 Pod 的分布如下：

node-1	node-2	node-3
pod-a *available*	pod-b *available*	pod-c *available*
pod-x *available*		

所有的3个 pod 都是 Deployment 中的一部分，并且它们共同拥有一个 PDB，要求至少有3个 pod 中的2个始终处于可用状态。

例如，假设集群管理员想要重启系统，升级内核版本来修复内核中的错误。集群管理员首先使用 `kubectl drain` 命令尝试排除 `node-1` 。该工具试图驱逐 `pod-a` 和 `pod-x` 。这立即成功。两个 Pod 同时进入终止状态。这时的集群处于这种状态：

node-1 *draining*	node-2	node-3
pod-a *terminating*	pod-b *available*	pod-c *available*
pod-x *terminating*		

Deployment 注意到其中有一个 pod 处于正在终止，因此会创建了一个 `pod-d` 来替换。由于 `node-1` 被封锁（cordon），它落在另一个节点上。同时其它控制器也创建了 `pod-y` 作为 `pod-x` 的替代品。

（注意：对于 `StatefulSet` ，`pod-a` 将被称为 `pod-1`，需要在替换之前完全终止，替代它的也称为 `pod-1`，但是具有不同的 UID，可以创建。否则，示例也适用于 StatefulSet。）

当前集群的状态如下：

node-1 *draining*	node-2	node-3
pod-a *terminating*	pod-b *available*	pod-c *available*
pod-x *terminating*	pod-d *starting*	pod-y

在某一时刻，pod 被终止，集群看起来像下面这样子：

node-1 *drained*	node-2	node-3
	pod-b *available*	pod-c *available*
	pod-d *starting*	pod-y

此时，如果一个急躁的集群管理员试图排空（drain）`node-2` 或 `node-3` ，drain 命令将被阻塞，因为对于 Deployment 只有2个可用的 pod，并且其 PDB 至少需要2个。经过一段时间，`pod-d` 变得可用。

node-1 *drained*	node-2	node-3
	pod-b *available*	pod-c *available*
	pod-d *available*	pod-y

现在，集群管理员尝试排空 `node-2` 。drain 命令将尝试按照某种顺序驱逐两个 pod，假设先是 `pod-b` ，然后再 `pod-d` 。它将成功驱逐 `pod-b` 。但是，当它试图驱逐 `pod-d` 时，将被拒绝，因为这样对 Deployment 来说将只剩下一个可用的 pod。

Deployment 将创建一个名为 `pod-e` 的 `pod-b` 的替代品。但是，集群中没有足够的资源来安排 `pod-e` 。那么，drain 命令就会被阻塞。集群最终可能是这种状态：

node-1 *drained*	node-2	node-3	*no node*
	pod-b *available*	pod-c *available*	pod-e *pending*
	pod-d *available*	pod-y	

此时，集群管理员需要向集群中添加回一个节点以继续升级操作。

您可以看到 Kubernetes 如何改变中断发生的速率，根据：

- 应用程序需要多少副本
- 正常关闭实例需要多长时间
- 启动新实例需要多长时间
- 控制器的类型
- 集群的资源能力

分离集群所有者和应用程序所有者角色

将集群管理者和应用程序所有者视为彼此知识有限的独立角色通常是很有用的。这种责任分离在这些情况下可能是有意义的：

- 当有许多应用程序团队共享一个 Kubernetes 集群，并且有自然的专业角色
- 使用第三方工具或服务来自动化集群管理

Pod Disruption Budget（Pod 中断预算） 通过在角色之间提供接口来支持这种角色分离。

如果您的组织中没有这样的职责分离，则可能不需要使用 Pod 中断预算。

如何在集群上执行中断操作

如果您是集群管理员，要对集群的所有节点执行中断操作，例如节点或系统软件升级，则可以使用以下选择：

- 在升级期间接受停机时间。
- 故障转移到另一个完整的副本集群。

    - 没有停机时间，但是对于重复的节点和人工协调成本可能是昂贵的。
- 编写可容忍中断的应用程序和使用 PDB。
    - 没有停机时间。
    - 最小的资源重复。
    - 允许更多的集群管理自动化。
    - 编写可容忍中断的应用程序是很棘手的，但对于可容忍自愿中断，和支持自动调整以容忍非自愿中断，两者在工作上有大量的重叠。

参考

- Disruptions - kubernetes.io
- 通过配置Pod Disruption Budget（Pod 中断预算）来执行保护应用程序的步骤。
- 了解更多关于排空节点的信息。

集群信息

为了管理异构和不同配置的主机，为了便于Pod的运维管理，Kubernetes中提供了很多集群管理的配置和管理功能，通过namespace划分的空间，通过为node节点创建label和taint用于pod的调度等。

Node

Node是kubernetes集群的工作节点，可以是物理机也可以是虚拟机。

Node的状态

Node包括如下状态信息：

- Address
  - HostName：可以被kubelet中的 `--hostname-override` 参数替代。
  - ExternalIP：可以被集群外部路由到的IP地址。
  - InternalIP：集群内部使用的IP，集群外部无法访问。
- Condition
  - OutOfDisk：磁盘空间不足时为 `True`
  - Ready：Node controller 40秒内没有收到node的状态报告为 `Unknown` ，健康为 `True` ，否则为 `False` 。
  - MemoryPressure：当node没有内存压力时为 `True` ，否则为 `False` 。
  - DiskPressure：当node没有磁盘压力时为 `True` ，否则为 `False` 。
- Capacity
  - CPU
  - 内存
  - 可运行的最大Pod个数
- Info：节点的一些版本信息，如OS、kubernetes、docker等

Node管理

禁止pod调度到该节点上

```
kubectl cordon <node>
```

驱逐该节点上的所有pod

```
kubectl drain <node>
```

该命令会删除该节点上的所有Pod（DaemonSet除外），在其他node上重新启动它们，通常该节点需要维护时使用该命令。直接使用该命令会自动调用 `kubectl cordon <node>` 命令。当该节点维护完成，启动了kubelet后，再使用 `kubectl uncordon <node>` 即可将该节点添加到kubernetes集群中。

Namespace

在一个Kubernetes集群中可以使用namespace创建多个“虚拟集群”，这些namespace之间可以完全隔离，也可以通过某种方式，让一个namespace中的service可以访问到其他的namespace中的服务，我们在CentOS中部署kubernetes1.6集群的时候就用到了好几个跨越namespace的服务，比如Traefik ingress和 `kube-system` namespace下的service就可以为整个集群提供服务，这些都需要通过RBAC定义集群级别的角色来实现。

哪些情况下适合使用多个namespace

因为namespace可以提供独立的命名空间，因此可以实现部分的环境隔离。当你的项目和人员众多的时候可以考虑根据项目属性，例如生产、测试、开发划分不同的namespace。

Namespace使用

获取集群中有哪些namespace

```
kubectl get ns
```

集群中默认会有 `default` 和 `kube-system` 这两个namespace。

在执行 `kubectl` 命令时可以使用 `-n` 指定操作的namespace。

用户的普通应用默认是在 `default` 下，与集群管理相关的为整个集群提供服务的应用一般部署在 `kube-system` 的namespace下，例如我们在安装kubernetes集群时部署的 `kubedns` 、 `heapseter` 、 `EFK` 等都是在这个namespace下面。

另外，并不是所有的资源对象都会对应namespace， `node` 和 `persistentVolume` 就不属于任何namespace。

Label

Label是附着到object上（例如Pod）的键值对。可以在创建object的时候指定，也可以在object创建后随时指定。Labels的值对系统本身并没有什么含义，只是对用户才有意义。

```
"labels": {
  "key1" : "value1",
  "key2" : "value2"
}
```

Kubernetes最终将对labels最终索引和反向索引用来优化查询和watch，在UI和命令行中会对它们排序。不要在label中使用大型、非标识的结构化数据，记录这样的数据应该用annotation。

动机

Label能够将组织架构映射到系统架构上（就像是康威定律），这样能够更便于微服务的管理，你可以给object打上如下类型的label：

- `"release" : "stable"` , `"release" : "canary"`
- `"environment" : "dev"` , `"environment" : "qa"` , `"environment" : "production"`
- `"tier" : "frontend"` , `"tier" : "backend"` , `"tier" : "cache"`
- `"partition" : "customerA"` , `"partition" : "customerB"`
- `"track" : "daily"` , `"track" : "weekly"`
- `"team" : "teamA"` , `"team:" : "teamB"`

语法和字符集

Label key的组成：

- 不得超过63个字符
- 可以使用前缀，使用/分隔，前缀必须是DNS子域，不得超过253个字符，系统中的自动化组件创建的label必须指定前缀， `kubernetes.io/` 由kubernetes

保留

- 起始必须是字母（大小写都可以）或数字，中间可以有连字符、下划线和点

Label value的组成：

- 不得超过63个字符
- 起始必须是字母（大小写都可以）或数字，中间可以有连字符、下划线和点

Label selector

Label不是唯一的，很多object可能有相同的label。

通过label selector，客户端／用户可以指定一个object集合，通过label selector对object的集合进行操作。

Label selector有两种类型：

- *equality-based* ：可以使用 `=` 、 `==` 、 `!=` 操作符，可以使用逗号分隔多个表达式
- *set-based* ：可以使用 `in` 、 `notin` 、 `!` 操作符，另外还可以没有操作符，直接写出某个label的key，表示过滤有某个key的object而不管该key的value是何值， `!` 表示没有该label的object

示例

```
$ kubectl get pods -l environment=production,tier=frontend
$ kubectl get pods -l 'environment in (production),tier in (frontend)'
$ kubectl get pods -l 'environment in (production, qa)'
$ kubectl get pods -l 'environment,environment notin (frontend)'
```

在API object中设置label selector

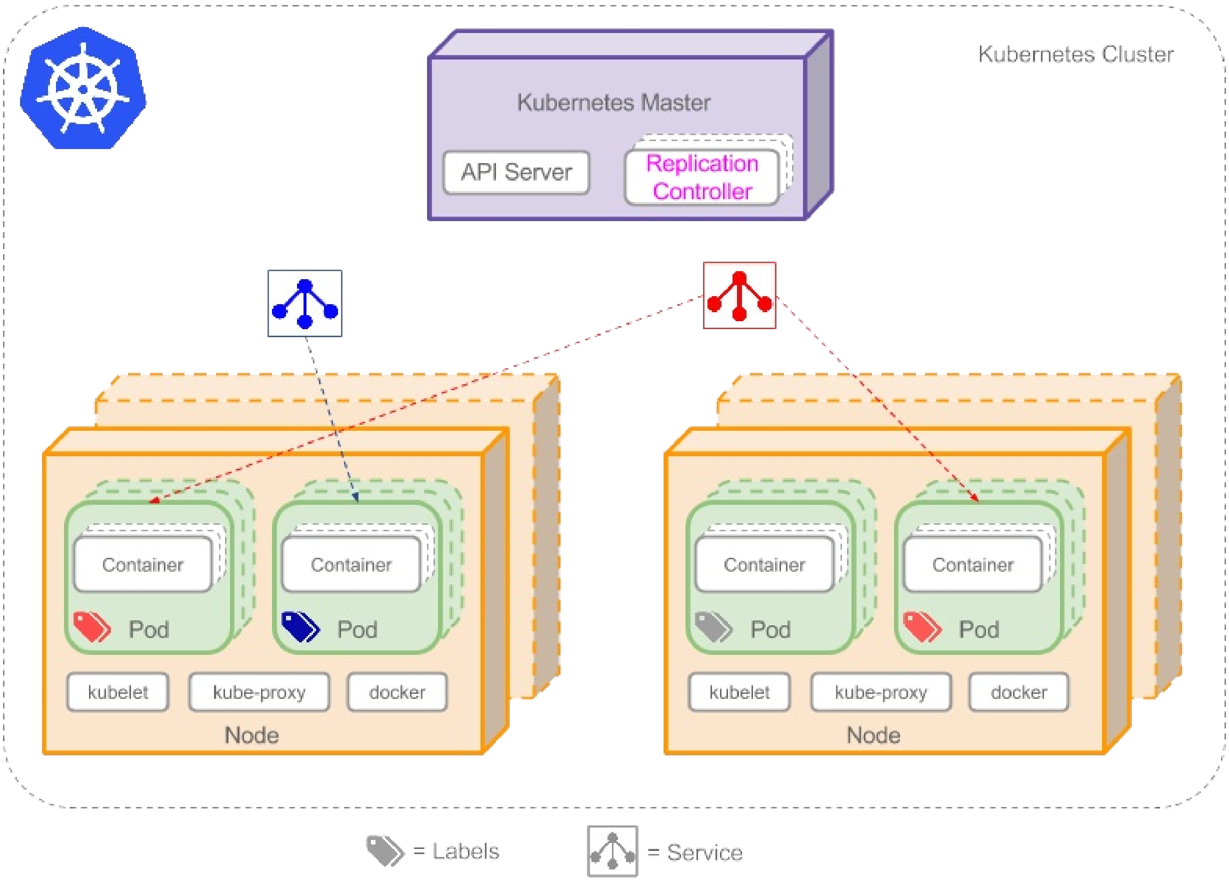
在 `service` 、 `replicationcontroller` 等object中有对pod的label selector，使用方法只能使用等于操作，例如：

```
selector:
    component: redis
```

在 `Job` 、 `Deployment` 、 `ReplicaSet` 和 `DaemonSet` 这些object中，支持*set-based*的过滤，例如：

```
selector:
  matchLabels:
    component: redis
  matchExpressions:
    - {key: tier, operator: In, values: [cache]}
    - {key: environment, operator: NotIn, values: [dev]}
```

如Service通过label selector将同一类型的pod作为一个服务expose出来。



图片 - *label*示意图

另外在node affinity和pod affinity中的label selector的语法又有些许不同，示例如下：

```
affinity:
  nodeAffinity:
    requiredDuringSchedulingIgnoredDuringExecution:
      nodeSelectorTerms:
      - matchExpressions:
        - key: kubernetes.io/e2e-az-name
          operator: In
          values:
          - e2e-az1
          - e2e-az2
    preferredDuringSchedulingIgnoredDuringExecution:
    - weight: 1
      preference:
        matchExpressions:
        - key: another-node-label-key
          operator: In
          values:
          - another-node-label-value
```

详见node selection。

Annotation

Annotation，顾名思义，就是注解。Annotation可以将Kubernetes资源对象关联到任意的非标识性元数据。使用客户端（如工具和库）可以检索到这些元数据。

关联元数据到对象

Label和Annotation都可以将元数据关联到Kubernetes资源对象。Label主要用于选择对象，可以挑选出满足特定条件的对象。相比之下，annotation 不能用于标识及选择对象。annotation中的元数据可多可少，可以是结构化的或非结构化的，也可以包含label中不允许出现的字符。

annotation和label一样都是key/value键值对映射结构：

```
"annotations": {
  "key1" : "value1",
  "key2" : "value2"
}
```

以下列出了一些可以记录在 annotation 中的对象信息：

- 声明配置层管理的字段。使用annotation关联这类字段可以用于区分以下几种配置来源：客户端或服务器设置的默认值，自动生成的字段或自动生成的 auto-scaling 和 auto-sizing 系统配置的字段。
- 创建信息、版本信息或镜像信息。例如时间戳、版本号、git分支、PR序号、镜像哈希值以及仓库地址。
- 记录日志、监控、分析或审计存储仓库的指针
- 可以用于debug的客户端（库或工具）信息，例如名称、版本和创建信息。
- 用户信息，以及工具或系统来源信息、例如来自非Kubernetes生态的相关对象的URL信息。
- 轻量级部署工具元数据，例如配置或检查点。
- 负责人的电话或联系方式，或能找到相关信息的目录条目信息，例如团队网站。

如果不使用annotation，您也可以将以上类型的信息存放在外部数据库或目录中，但这样做不利于创建用于部署、管理、内部检查的共享工具和客户端库。

示例

如 Istio 的 Deployment 配置中就使用到了 annotation：

```yaml
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: istio-manager
spec:
  replicas: 1
  template:
    metadata:
      annotations:
        alpha.istio.io/sidecar: ignore
      labels:
        istio: manager
    spec:
      serviceAccountName: istio-manager-service-account
      containers:
      - name: discovery
        image: harbor-001.jimmysong.io/library/manager:0.1.5
        imagePullPolicy: Always
        args: ["discovery", "-v", "2"]
        ports:
        - containerPort: 8080
        env:
        - name: POD_NAMESPACE
          valueFrom:
            fieldRef:
              apiVersion: v1
              fieldPath: metadata.namespace
      - name: apiserver
        image: harbor-001.jimmysong.io/library/manager:0.1.5
        imagePullPolicy: Always
        args: ["apiserver", "-v", "2"]
        ports:
        - containerPort: 8081
        env:
        - name: POD_NAMESPACE
          valueFrom:
            fieldRef:
              apiVersion: v1
              fieldPath: metadata.namespace
```

`alpha.istio.io/sidecar` 注解就是用来控制是否自动向 pod 中注入 sidecar 的。参考：安装 Istio sidecar - istio.doczh.cn。

GitbookUpdated at 2018-05-09 14:15:47

Taint和Toleration（污点和容忍）

Taint（污点）和 Toleration（容忍）可以作用于 node 和 pod 上，其目的是优化 pod 在集群间的调度，这跟节点亲和性类似，只不过它们作用的方式相反，具有 taint 的 node 和 pod 是互斥关系，而具有节点亲和性关系的 node 和 pod 是相吸的。另外还有可以给 node 节点设置 label，通过给 pod 设置 `nodeSelector` 将 pod 调度到具有匹配标签的节点上。

Taint 和 toleration 相互配合，可以用来避免 pod 被分配到不合适的节点上。每个节点上都可以应用一个或多个 taint ，这表示对于那些不能容忍这些 taint 的 pod，是不会被该节点接受的。如果将 toleration 应用于 pod 上，则表示这些 pod 可以（但不要求）被调度到具有相应 taint 的节点上。

示例

以下分别以为 node 设置 taint 和为 pod 设置 toleration 为例。

为 node 设置 taint

为 node1 设置 taint：

```
kubectl taint nodes node1 key1=value1:NoSchedule
kubectl taint nodes node1 key1=value1:NoExecute
kubectl taint nodes node1 key2=value2:NoSchedule
```

删除上面的 taint：

```
kubectl taint nodes node1 key1:NoSchedule-
kubectl taint nodes node1 key1:NoExecute-
kubectl taint nodes node1 key2:NoSchedule-
```

查看 node1 上的 taint：

```
kubectl describe nodes node1
```

为 pod 设置 toleration

只要在 pod 的 spec 中设置 tolerations 字段即可，可以有多个 `key` ，如下所示：

```
tolerations:
- key: "key1"
  operator: "Equal"
  value: "value1"
  effect: "NoSchedule"
- key: "key1"
  operator: "Equal"
  value: "value1"
  effect: "NoExecute"
- key: "node.alpha.kubernetes.io/unreachable"
  operator: "Exists"
  effect: "NoExecute"
  tolerationSeconds: 6000
```

- `value` 的值可以为 `NoSchedule` 、 `PreferNoSchedule` 或 `NoExecute` 。
- `tolerationSeconds` 是当 pod 需要被驱逐时，可以继续在 node 上运行的时间。

详细使用方法请参考官方文档。

参考

- Taints and Tolerations - kuberentes.io

垃圾收集

Kubernetes 垃圾收集器的角色是删除指定的对象，这些对象曾经有但以后不再拥有 Owner 了。

注意：垃圾收集是 beta 特性，在 Kubernetes 1.4 及以上版本默认启用。

Owner 和 Dependent

一些 Kubernetes 对象是其它一些的 Owner。例如，一个 ReplicaSet 是一组 Pod 的 Owner。具有 Owner 的对象被称为是 Owner 的 *Dependent*。每个 Dependent 对象具有一个指向其所属对象的 `metadata.ownerReferences` 字段。

有时，Kubernetes 会自动设置 `ownerReference` 的值。例如，当创建一个 ReplicaSet 时，Kubernetes 自动设置 ReplicaSet 中每个 Pod 的 `ownerReference` 字段值。在 1.6 版本，Kubernetes 会自动为一些对象设置 `ownerReference` 的值，这些对象是由 ReplicationController、ReplicaSet、StatefulSet、DaemonSet 和 Deployment 所创建或管理。

也可以通过手动设置 `ownerReference` 的值，来指定 Owner 和 Dependent 之间的关系。

这有一个配置文件，表示一个具有 3 个 Pod 的 ReplicaSet：

```yaml
apiVersion: extensions/v1beta1
kind: ReplicaSet
metadata:
  name: my-repset
spec:
  replicas: 3
  selector:
    matchLabels:
      pod-is-for: garbage-collection-example
  template:
    metadata:
      labels:
        pod-is-for: garbage-collection-example
    spec:
      containers:
      - name: nginx
        image: nginx
```

如果创建该 ReplicaSet，然后查看 Pod 的 metadata 字段，能够看到 OwnerReferences 字段：

```
kubectl create -f https://k8s.io/docs/concepts/abstractions/controllers/my-repset.yaml
kubectl get pods --output=yaml
```

输出显示了 Pod 的 Owner 是名为 my-repset 的 ReplicaSet：

```yaml
apiVersion: v1
kind: Pod
metadata:
  ...
  ownerReferences:
  - apiVersion: extensions/v1beta1
    controller: true
    blockOwnerDeletion: true
    kind: ReplicaSet
    name: my-repset
    uid: d9607e19-f88f-11e6-a518-42010a800195
  ...
```

控制垃圾收集器删除 Dependent

当删除对象时，可以指定是否该对象的 Dependent 也自动删除掉。自动删除 Dependent 也称为 级联删除。Kubernetes 中有两种 级联删除 的模式：*background* 模式和 *foreground* 模式。

如果删除对象时，不自动删除它的 Dependent，这些 Dependent 被称作是原对象的 孤儿。

Background 级联删除

在 *background* 级联删除 模式下，Kubernetes 会立即删除 Owner 对象，然后垃圾收集器会在后台删除这些 Dependent。

Foreground 级联删除

在 *foreground* 级联删除 模式下，根对象首先进入 "删除中" 状态。在 "删除中" 状态会有如下的情况：

- 对象仍然可以通过 REST API 可见
- 会设置对象的 `deletionTimestamp` 字段
- 对象的 `metadata.finalizers` 字段包含了值 “foregroundDeletion”

一旦被设置为 “删除中” 状态，垃圾收集器会删除对象的所有 Dependent。垃圾收集器删除了所有 “Blocking” 的 Dependent（对象的 `ownerReference.blockOwnerDeletion=true` ）之后，它会删除 Owner 对象。

注意，在 “foreground 删除” 模式下，Dependent 只有通过 `ownerReference.blockOwnerDeletion` 才能阻止删除 Owner 对象。在 Kubernetes 1.7 版本中将增加 admission controller，基于 Owner 对象上的删除权限来控制用户去设置 `blockOwnerDeletion` 的值为 true，所以未授权的 Dependent 不能够延迟 Owner 对象的删除。

如果一个对象的 `ownerReferences` 字段被一个 Controller（例如 Deployment 或 ReplicaSet）设置， `blockOwnerDeletion` 会被自动设置，没必要手动修改这个字段。

设置级联删除策略

通过为 Owner 对象设置 `deleteOptions.propagationPolicy` 字段，可以控制级联删除策略。可能的取值包括：“orphan”、“Foreground” 或 “Background”。

对很多 Controller 资源，包括 ReplicationController、ReplicaSet、StatefulSet、DaemonSet 和 Deployment，默认的垃圾收集策略是 `orphan` 。因此，除非指定其它的垃圾收集策略，否则所有 Dependent 对象使用的都是 `orphan` 策略。

下面是一个在后台删除 Dependent 对象的例子：

```
kubectl proxy --port=8080
curl -X DELETE localhost:8080/apis/extensions/v1beta1/namespaces
/default/replicasets/my-repset \
-d '{"kind":"DeleteOptions","apiVersion":"v1","propagationPolicy
":"Background"}' \
-H "Content-Type: application/json"
```

下面是一个在前台删除 Dependent 对象的例子：

```
kubectl proxy --port=8080
curl -X DELETE localhost:8080/apis/extensions/v1beta1/namespaces
/default/replicasets/my-repset \
-d '{"kind":"DeleteOptions","apiVersion":"v1","propagationPolicy
":"Foreground"}' \
-H "Content-Type: application/json"
```

下面是一个孤儿 Dependent 的例子：

```
kubectl proxy --port=8080
curl -X DELETE localhost:8080/apis/extensions/v1beta1/namespaces
/default/replicasets/my-repset \
-d '{"kind":"DeleteOptions","apiVersion":"v1","propagationPolicy
":"Orphan"}' \
-H "Content-Type: application/json"
```

kubectl 也支持级联删除。 通过设置 `--cascade` 为 true，可以使用 kubectl 自动删除 Dependent 对象。设置 `--cascade` 为 false，会使 Dependent 对象成为孤儿 Dependent 对象。 `--cascade` 的默认值是 true。

下面是一个例子，使一个 ReplicaSet 的 Dependent 对象成为孤儿 Dependent：

```
kubectl delete replicaset my-repset --cascade=false
```

已知的问题

- 1.7 版本，垃圾收集不支持 自定义资源，比如那些通过 CustomResourceDefinition 新增，或者通过 API server 聚集而成的资源对象。

其它已知的问题

原文地址：https://k8smeetup.github.io/docs/concepts/workloads/controllers/garbage-collection/

译者：shirdrn

控制器

Kubernetes中内建了很多controller（控制器），这些相当于一个状态机，用来控制Pod的具体状态和行为。

Deployment

简述

Deployment 为 Pod 和 ReplicaSet 提供了一个声明式定义(declarative)方法，用来替代以前的ReplicationController 来方便的管理应用。典型的应用场景包括：

- 定义Deployment来创建Pod和ReplicaSet
- 滚动升级和回滚应用
- 扩容和缩容
- 暂停和继续Deployment

比如一个简单的nginx应用可以定义为

```yaml
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 3
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
        - containerPort: 80
```

扩容：

```
kubectl scale deployment nginx-deployment --replicas 10
```

如果集群支持 horizontal pod autoscaling 的话，还可以为Deployment设置自动扩展：

```
kubectl autoscale deployment nginx-deployment --min=10 --max=15
--cpu-percent=80
```

更新镜像也比较简单:

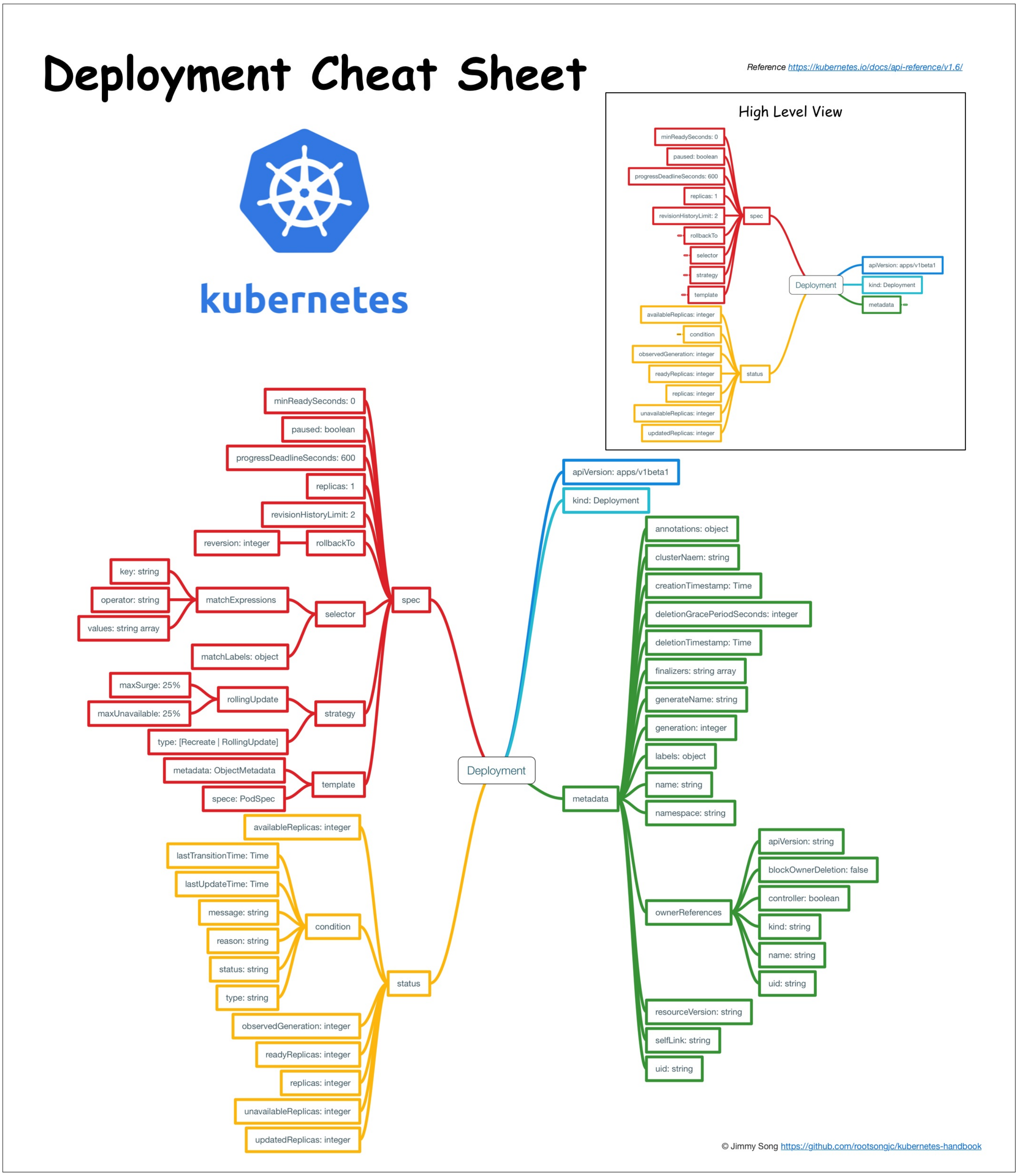
```
kubectl set image deployment/nginx-deployment nginx=nginx:1.9.1
```

回滚：

```
kubectl rollout undo deployment/nginx-deployment
```

Deployment 结构示意图

参考：https://kubernetes.io/docs/api-reference/v1.6/#deploymentspec-v1beta1-apps



图片 - *kubernetes deployment cheatsheet*

Deployment 概念详细解析

本文翻译自kubernetes官方文档：https://kubernetes.io/docs/concepts/workloads/controllers/deployment.md

根据2017年5月10日的Commit 8481c02 翻译。

Deployment 是什么？

Deployment为Pod和Replica Set（下一代Replication Controller）提供声明式更新。

您只需要在 Deployment 中描述您想要的目标状态是什么，Deployment controller 就会帮您将 Pod 和ReplicaSet 的实际状态改变到您的目标状态。您可以定义一个全新的 Deployment 来创建 ReplicaSet 或者删除已有的 Deployment 并创建一个新的来替换。

注意：您不该手动管理由 Deployment 创建的 ReplicaSet，否则您就篡越了 Deployment controller 的职责！下文罗列了 Deployment 对象中已经覆盖了所有的用例。如果未有覆盖您所有需要的用例，请直接在 Kubernetes 的代码库中提issue。

典型的用例如下：

- 使用Deployment来创建ReplicaSet。ReplicaSet在后台创建pod。检查启动状态，看它是成功还是失败。
- 然后，通过更新Deployment的PodTemplateSpec字段来声明Pod的新状态。这会创建一个新的ReplicaSet，Deployment会按照控制的速率将pod从旧的ReplicaSet移动到新的ReplicaSet中。
- 如果当前状态不稳定，回滚到之前的Deployment revision。每次回滚都会更新Deployment的revision。
- 扩容Deployment以满足更高的负载。
- 暂停Deployment来应用PodTemplateSpec的多个修复，然后恢复上线。
- 根据Deployment 的状态判断上线是否hang住了。
- 清除旧的不必要的 ReplicaSet。

创建 Deployment

下面是一个 Deployment 示例，它创建了一个 ReplicaSet 来启动3个 nginx pod。

下载示例文件并执行命令：

```
$ kubectl create -f https://kubernetes.io/docs/user-guide/nginx-
deployment.yaml --record
deployment "nginx-deployment" created
```

将kubectl的 `--record` 的 flag 设置为 `true` 可以在 annotation 中记录当前命令创建或者升级了该资源。这在未来会很有用，例如，查看在每个 Deployment revision 中执行了哪些命令。

然后立即执行 `get` 将获得如下结果：

```
$ kubectl get deployments
NAME               DESIRED   CURRENT   UP-TO-DATE   AVAILABLE
AGE
nginx-deployment   3         0         0            0
1s
```

输出结果表明我们希望的repalica数是3（根据deployment中的 `.spec.replicas` 配置）当前replica数（ `.status.replicas` ）是0, 最新的replica数（ `.status.updatedReplicas` ）是0，可用的replica数（ `.status.availableReplicas` ）是0。

过几秒后再执行 `get` 命令，将获得如下输出：

```
$ kubectl get deployments
NAME               DESIRED   CURRENT   UP-TO-DATE   AVAILABLE
AGE
nginx-deployment   3         3         3            3
18s
```

我们可以看到Deployment已经创建了3个 replica，所有的 replica 都已经是最新的了（包含最新的pod template），可用的（根据Deployment中的 `.spec.minReadySeconds` 声明，处于已就绪状态的pod的最少个数）。执行 `kubectl get rs` 和 `kubectl get pods` 会显示Replica Set（RS）和Pod已创建。

```
$ kubectl get rs
NAME                          DESIRED   CURRENT   READY     AGE
nginx-deployment-2035384211   3         3         0         18s
```

您可能会注意到 ReplicaSet 的名字总是 `<Deployment的名字>-<pod template的hash值>` 。

```
$ kubectl get pods --show-labels
NAME                                READY     STATUS    RESTARTS
   AGE       LABELS
nginx-deployment-2035384211-7ci7o   1/1       Running   0
   18s       app=nginx,pod-template-hash=2035384211
nginx-deployment-2035384211-kzszj   1/1       Running   0
   18s       app=nginx,pod-template-hash=2035384211
nginx-deployment-2035384211-qqcnn   1/1       Running   0
   18s       app=nginx,pod-template-hash=2035384211
```

刚创建的Replica Set将保证总是有3个 nginx 的 pod 存在。

注意： 您必须在 Deployment 中的 selector 指定正确的 pod template label（在该示例中是 `app = nginx` ），不要跟其他的 controller 的 selector 中指定的 pod template label 搞混了（包括 Deployment、Replica Set、Replication Controller 等）。**Kubernetes** 本身并不会阻止您任意指定 **pod template label** ，但是如果您真的这么做了，这些 controller 之间会相互打架，并可能导致不正确的行为。

Pod-template-hash label

注意：这个 label 不是用户指定的！

注意上面示例输出中的 pod label 里的 pod-template-hash label。当 Deployment 创建或者接管 ReplicaSet 时，Deployment controller 会自动为 Pod 添加 pod-template-hash label。这样做的目的是防止 Deployment 的子ReplicaSet 的 pod 名字重复。通过将 ReplicaSet 的 PodTemplate 进行哈希散列，使用生成的哈希值作为 label 的值，并添加到 ReplicaSet selector 里、 pod template label 和 ReplicaSet 管理中的 Pod 上。

更新Deployment

注意： Deployment 的 rollout 当且仅当 Deployment 的 pod template（例如 `.spec.template` ）中的label更新或者镜像更改时被触发。其他更新，例如扩容Deployment不会触发 rollout。

假如我们现在想要让 nginx pod 使用 `nginx:1.9.1` 的镜像来代替原来的 `nginx:1.7.9` 的镜像。

```
$ kubectl set image deployment/nginx-deployment nginx=nginx:1.9.
1
deployment "nginx-deployment" image updated
```

我们可以使用 `edit` 命令来编辑 Deployment，修改 `.spec.template.spec.containers[0].image` ，将 `nginx:1.7.9` 改写成 `nginx:1.9.1` 。

```
$ kubectl edit deployment/nginx-deployment
deployment "nginx-deployment" edited
```

查看 rollout 的状态，只要执行：

```
$ kubectl rollout status deployment/nginx-deployment
Waiting for rollout to finish: 2 out of 3 new replicas have been
 updated...
deployment "nginx-deployment" successfully rolled out
```

Rollout 成功后， `get` Deployment：

```
$ kubectl get deployments
NAME               DESIRED   CURRENT   UP-TO-DATE   AVAILABLE
AGE
nginx-deployment   3         3         3            3
36s
```

UP-TO-DATE 的 replica 的数目已经达到了配置中要求的数目。

CURRENT 的 replica 数表示 Deployment 管理的 replica 数量，AVAILABLE 的 replica 数是当前可用的replica数量。

我们通过执行 `kubectl get rs` 可以看到 Deployment 更新了Pod，通过创建一个新的 ReplicaSet 并扩容了3个 replica，同时将原来的 ReplicaSet 缩容到了0个 replica。

```
$ kubectl get rs
NAME                          DESIRED   CURRENT   READY   AGE
nginx-deployment-1564180365   3         3         0       6s
nginx-deployment-2035384211   0         0         0       36s
```

执行 `get pods` 只会看到当前的新的 pod:

```
$ kubectl get pods
NAME                                READY     STATUS    RESTARTS
   AGE
nginx-deployment-1564180365-khku8   1/1       Running   0
   14s
nginx-deployment-1564180365-nacti   1/1       Running   0
   14s
nginx-deployment-1564180365-z9gth   1/1       Running   0
   14s
```

下次更新这些 pod 的时候，只需要更新 Deployment 中的 pod 的 template 即可。

Deployment 可以保证在升级时只有一定数量的 Pod 是 down 的。默认的，它会确保至少有比期望的Pod数量少一个是up状态（最多一个不可用）。

Deployment 同时也可以确保只创建出超过期望数量的一定数量的 Pod。默认的，它会确保最多比期望的Pod数量多一个的 Pod 是 up 的（最多1个 surge ）。

在未来的 **Kuberentes** 版本中，将从**1-1**变成**25%-25%**。

例如，如果您自己看下上面的 Deployment，您会发现，开始创建一个新的 Pod，然后删除一些旧的 Pod 再创建一个新的。当新的Pod创建出来之前不会杀掉旧的 Pod。这样能够确保可用的 Pod 数量至少有2个，Pod的总数最多4个。

```
$ kubectl describe deployments
Name:           nginx-deployment
Namespace:      default
CreationTimestamp:  Tue, 15 Mar 2016 12:01:06 -0700
Labels:         app=nginx
Selector:       app=nginx
Replicas:       3 updated | 3 total | 3 available | 0 unavailabl
e
StrategyType:       RollingUpdate
MinReadySeconds:    0
RollingUpdateStrategy:  1 max unavailable, 1 max surge
OldReplicaSets:     <none>
NewReplicaSet:      nginx-deployment-1564180365 (3/3 replicas cr
eated)
Events:
  FirstSeen LastSeen    Count   From                     Subobje
ctPath   Type        Reason              Message
  --------- --------    -----   ----                     -------
------   --------    ------              -------
  36s       36s         1       {deployment-controller }
         Normal      ScalingReplicaSet   Scaled up replica set n
ginx-deployment-2035384211 to 3
  23s       23s         1       {deployment-controller }
         Normal      ScalingReplicaSet   Scaled up replica set n
ginx-deployment-1564180365 to 1
  23s       23s         1       {deployment-controller }
         Normal      ScalingReplicaSet   Scaled down replica set
 nginx-deployment-2035384211 to 2
  23s       23s         1       {deployment-controller }
         Normal      ScalingReplicaSet   Scaled up replica set n
ginx-deployment-1564180365 to 2
  21s       21s         1       {deployment-controller }
         Normal      ScalingReplicaSet   Scaled down replica set
 nginx-deployment-2035384211 to 0
  21s       21s         1       {deployment-controller }
         Normal      ScalingReplicaSet   Scaled up replica set n
ginx-deployment-1564180365 to 3
```

我们可以看到当我们刚开始创建这个 Deployment 的时候，创建了一个 ReplicaSet（nginx-deployment-2035384211），并直接扩容到了3个 replica。

当我们更新这个 Deployment 的时候，它会创建一个新的 ReplicaSet（nginx-deployment-1564180365），将它扩容到1个replica，然后缩容原先的 ReplicaSet 到2个 replica，此时满足至少2个 Pod 是可用状态，同一时刻最多有4个 Pod 处于创建的状态。

接着继续使用相同的 rolling update 策略扩容新的 ReplicaSet 和缩容旧的 ReplicaSet。最终，将会在新的 ReplicaSet 中有3个可用的 replica，旧的 ReplicaSet 的 replica 数目变成0。

Rollover（多个rollout并行）

每当 Deployment controller 观测到有新的 deployment 被创建时，如果没有已存在的 ReplicaSet 来创建期望个数的 Pod 的话，就会创建出一个新的 ReplicaSet 来做这件事。已存在的 ReplicaSet 控制 label 与 `.spec.selector` 匹配但是 template 跟 `.spec.template` 不匹配的 Pod 缩容。最终，新的 ReplicaSet 将会扩容出 `.spec.replicas` 指定数目的 Pod，旧的 ReplicaSet 会缩容到0。

如果您更新了一个的已存在并正在进行中的 Deployment，每次更新 Deployment都会创建一个新的 ReplicaSet并扩容它，同时回滚之前扩容的 ReplicaSet ——将它添加到旧的 ReplicaSet 列表中，开始缩容。

例如，假如您创建了一个有5个 `niginx:1.7.9` replica的 Deployment，但是当还只有3个 `nginx:1.7.9` 的 replica 创建出来的时候您就开始更新含有5个 `nginx:1.9.1` replica 的 Deployment。在这种情况下，Deployment 会立即杀掉已创建的3个 `nginx:1.7.9` 的 Pod，并开始创建 `nginx:1.9.1` 的 Pod。它不会等到所有的5个 `nginx:1.7.9` 的 Pod 都创建完成后才开始改变航道。

Label selector 更新

我们通常不鼓励更新 label selector，我们建议事先规划好您的 selector。

任何情况下，只要您想要执行 label selector 的更新，请一定要谨慎并确认您已经预料到所有可能因此导致的后果。

- 增添 selector 需要同时在 Deployment 的 spec 中更新新的 label，否则将返回校验错误。此更改是不可覆盖的，这意味着新的 selector 不会选择使用旧 selector 创建的 ReplicaSet 和 Pod，从而导致所有旧版本的 ReplicaSet 都被丢弃，并创建新的 ReplicaSet。
- 更新 selector，即更改 selector key 的当前值，将导致跟增添 selector 同样的后果。
- 删除 selector，即删除 Deployment selector 中的已有的 key，不需要对 Pod template label 做任何更改，现有的 ReplicaSet 也不会成为孤儿，但是请注意，删除的 label 仍然存在于现有的 Pod 和 ReplicaSet 中。

回退Deployment

有时候您可能想回退一个 Deployment，例如，当 Deployment 不稳定时，比如一直 crash looping。

默认情况下，kubernetes 会在系统中保存前两次的 Deployment 的 rollout 历史记录，以便您可以随时回退（您可以修改 `revision history limit` 来更改保存的revision数）。

注意： 只要 Deployment 的 rollout 被触发就会创建一个 revision。也就是说当且仅当 Deployment 的 Pod template（如 `.spec.template` ）被更改，例如更新template 中的 label 和容器镜像时，就会创建出一个新的 revision。

其他的更新，比如扩容 Deployment 不会创建 revision——因此我们可以很方便的手动或者自动扩容。这意味着当您回退到历史 revision 时，只有 Deployment 中的 Pod template 部分才会回退。

假设我们在更新 Deployment 的时候犯了一个拼写错误，将镜像的名字写成了 `nginx:1.91` ，而正确的名字应该是 `nginx:1.9.1` ：

```
$ kubectl set image deployment/nginx-deployment nginx=nginx:1.91
deployment "nginx-deployment" image updated
```

Rollout 将会卡住。

```
$ kubectl rollout status deployments nginx-deployment
Waiting for rollout to finish: 2 out of 3 new replicas have been
 updated...
```

按住 Ctrl-C 停止上面的 rollout 状态监控。

您会看到旧的 replica（nginx-deployment-1564180365 和 nginx-deployment-2035384211）和新的 replica （nginx-deployment-3066724191）数目都是2个。

```
$ kubectl get rs
NAME                          DESIRED    CURRENT    READY    AGE
nginx-deployment-1564180365   2          2          0        25s
nginx-deployment-2035384211   0          0          0        36s
nginx-deployment-3066724191   2          2          2        6s
```

看下创建 Pod，您会看到有两个新的 ReplicaSet 创建的 Pod 处于 ImagePullBackOff 状态，循环拉取镜像。

```
$ kubectl get pods
NAME                                READY     STATUS
 RESTARTS   AGE
nginx-deployment-1564180365-70iae   1/1       Running
 0          25s
nginx-deployment-1564180365-jbqqo   1/1       Running
 0          25s
nginx-deployment-3066724191-08mng   0/1       ImagePullBackOff
 0          6s
nginx-deployment-3066724191-eocby   0/1       ImagePullBackOff
 0          6s
```

注意，Deployment controller会自动停止坏的 rollout，并停止扩容新的 ReplicaSet。

```
$ kubectl describe deployment
Name:           nginx-deployment
Namespace:      default
CreationTimestamp:  Tue, 15 Mar 2016 14:48:04 -0700
Labels:         app=nginx
Selector:       app=nginx
Replicas:       2 updated | 3 total | 2 available | 2 unavailabl
e
StrategyType:       RollingUpdate
MinReadySeconds:    0
RollingUpdateStrategy:  1 max unavailable, 1 max surge
OldReplicaSets:     nginx-deployment-1564180365 (2/2 replicas cr
eated)
NewReplicaSet:      nginx-deployment-3066724191 (2/2 replicas cr
eated)
Events:
  FirstSeen LastSeen    Count   From                    Subobjec
tPath   Type        Reason              Message
  --------- --------    -----   ----                    --------
-----   --------    ------              -------
  1m        1m          1       {deployment-controller }
        Normal      ScalingReplicaSet   Scaled up replica set ng
inx-deployment-2035384211 to 3
  22s       22s         1       {deployment-controller }
        Normal      ScalingReplicaSet   Scaled up replica set ng
inx-deployment-1564180365 to 1
  22s       22s         1       {deployment-controller }
        Normal      ScalingReplicaSet   Scaled down replica set
nginx-deployment-2035384211 to 2
  22s       22s         1       {deployment-controller }
        Normal      ScalingReplicaSet   Scaled up replica set ng
inx-deployment-1564180365 to 2
  21s       21s         1       {deployment-controller }
        Normal      ScalingReplicaSet   Scaled down replica set
nginx-deployment-2035384211 to 0
  21s       21s         1       {deployment-controller }
        Normal      ScalingReplicaSet   Scaled up replica set ng
inx-deployment-1564180365 to 3
  13s       13s         1       {deployment-controller }
        Normal      ScalingReplicaSet   Scaled up replica set ng
inx-deployment-3066724191 to 1
  13s       13s         1       {deployment-controller }
        Normal      ScalingReplicaSet   Scaled down replica set
nginx-deployment-1564180365 to 2
  13s       13s         1       {deployment-controller }
        Normal      ScalingReplicaSet   Scaled up replica set ng
inx-deployment-3066724191 to 2
```

为了修复这个问题，我们需要回退到稳定的 Deployment revision。

检查 Deployment 升级的历史记录

首先，检查下 Deployment 的 revision：

```
$ kubectl rollout history deployment/nginx-deployment
deployments "nginx-deployment":
REVISION    CHANGE-CAUSE
1           kubectl create -f https://kubernetes.io/docs/user-gu
ide/nginx-deployment.yaml--record
2           kubectl set image deployment/nginx-deployment nginx=
nginx:1.9.1
3           kubectl set image deployment/nginx-deployment nginx=
nginx:1.91
```

因为我们创建 Deployment 的时候使用了 `--record` 参数可以记录命令，我们可以很方便的查看每次 revision 的变化。

查看单个revision 的详细信息：

```
$ kubectl rollout history deployment/nginx-deployment --revision=
2
deployments "nginx-deployment" revision 2
  Labels:       app=nginx
          pod-template-hash=1159050644
  Annotations:  kubernetes.io/change-cause=kubectl set image dep
loyment/nginx-deployment nginx=nginx:1.9.1
  Containers:
   nginx:
    Image:      nginx:1.9.1
    Port:       80/TCP
     QoS Tier:
        cpu:      BestEffort
        memory:   BestEffort
    Environment Variables:      <none>
  No volumes.
```

回退到历史版本

现在，我们可以决定回退当前的 rollout 到之前的版本：

```
$ kubectl rollout undo deployment/nginx-deployment
deployment "nginx-deployment" rolled back
```

也可以使用 `--revision` 参数指定某个历史版本：

```
$ kubectl rollout undo deployment/nginx-deployment --to-revision=
2
deployment "nginx-deployment" rolled back
```

与 rollout 相关的命令详细文档见kubectl rollout。

该 Deployment 现在已经回退到了先前的稳定版本。如您所见，Deployment controller产生了一个回退到revison 2的 `DeploymentRollback` 的 event。

```
$ kubectl get deployment
NAME               DESIRED    CURRENT    UP-TO-DATE    AVAILABLE
AGE
nginx-deployment   3          3          3             3
30m

$ kubectl describe deployment
Name:           nginx-deployment
Namespace:      default
CreationTimestamp:  Tue, 15 Mar 2016 14:48:04 -0700
Labels:         app=nginx
Selector:       app=nginx
Replicas:       3 updated | 3 total | 3 available | 0 unavailabl
e
StrategyType:       RollingUpdate
MinReadySeconds:    0
RollingUpdateStrategy:  1 max unavailable, 1 max surge
OldReplicaSets:     <none>
NewReplicaSet:      nginx-deployment-1564180365 (3/3 replicas cr
eated)
Events:
  FirstSeen LastSeen    Count   From                    Subobjec
tPath   Type        Reason              Message
  --------- --------    -----   ----                    --------
-----   --------    ------              -------
  30m       30m         1       {deployment-controller }
        Normal      ScalingReplicaSet   Scaled up replica set ng
inx-deployment-2035384211 to 3
  29m       29m         1       {deployment-controller }
        Normal      ScalingReplicaSet   Scaled up replica set ng
inx-deployment-1564180365 to 1
  29m       29m         1       {deployment-controller }
        Normal      ScalingReplicaSet   Scaled down replica set
nginx-deployment-2035384211 to 2
  29m       29m         1       {deployment-controller }
        Normal      ScalingReplicaSet   Scaled up replica set ng
inx-deployment-1564180365 to 2
```

```
  29m       29m         1       {deployment-controller }
        Normal      ScalingReplicaSet   Scaled down replica set
nginx-deployment-2035384211 to 0
  29m       29m         1       {deployment-controller }
        Normal      ScalingReplicaSet   Scaled up replica set ng
inx-deployment-3066724191 to 2
  29m       29m         1       {deployment-controller }
        Normal      ScalingReplicaSet   Scaled up replica set ng
inx-deployment-3066724191 to 1
  29m       29m         1       {deployment-controller }
        Normal      ScalingReplicaSet   Scaled down replica set
nginx-deployment-1564180365 to 2
  2m        2m          1       {deployment-controller }
        Normal      ScalingReplicaSet   Scaled down replica set
nginx-deployment-3066724191 to 0
  2m        2m          1       {deployment-controller }
        Normal      DeploymentRollback  Rolled back deployment "
nginx-deployment" to revision 2
  29m       2m          2       {deployment-controller }
        Normal      ScalingReplicaSet   Scaled up replica set ng
inx-deployment-1564180365 to 3
```

清理 Policy

您可以通过设置 `.spec.revisonHistoryLimit` 项来指定 deployment 最多保留多少 revision 历史记录。默认的会保留所有的 revision；如果将该项设置为0，Deployment就不允许回退了。

Deployment 扩容

您可以使用以下命令扩容 Deployment：

```
$ kubectl scale deployment nginx-deployment --replicas 10
deployment "nginx-deployment" scaled
```

假设您的集群中启用了horizontal pod autoscaling，您可以给 Deployment 设置一个 autoscaler，基于当前 Pod的 CPU 利用率选择最少和最多的 Pod 数。

```
$ kubectl autoscale deployment nginx-deployment --min=10 --max=1
5 --cpu-percent=80
deployment "nginx-deployment" autoscaled
```

比例扩容

RollingUpdate Deployment 支持同时运行一个应用的多个版本。或者 autoscaler 扩容 RollingUpdate Deployment 的时候，正在中途的 rollout（进行中或者已经暂停的），为了降低风险，Deployment controller 将会平衡已存在的活动中的 ReplicaSet（有 Pod 的 ReplicaSet）和新加入的 replica。这被称为比例扩容。

例如，您正在运行中含有10个 replica 的 Deployment。maxSurge=3，maxUnavailable=2。

```
$ kubectl get deploy
NAME                 DESIRED    CURRENT    UP-TO-DATE    AVAILABLE
  AGE
nginx-deployment     10         10         10            10
  50s
```

您更新了一个镜像，而在集群内部无法解析。

```
$ kubectl set image deploy/nginx-deployment nginx=nginx:sometag
deployment "nginx-deployment" image updated
```

镜像更新启动了一个包含ReplicaSet nginx-deployment-1989198191的新的rollout，但是它被阻塞了，因为我们上面提到的maxUnavailable。

```
$ kubectl get rs
NAME                          DESIRED    CURRENT    READY    AGE
nginx-deployment-1989198191   5          5          0        9s
nginx-deployment-618515232    8          8          8        1m
```

然后发起了一个新的Deployment扩容请求。autoscaler将Deployment的repllica数目增加到了15个。Deployment controller需要判断在哪里增加这5个新的replica。如果我们没有谁用比例扩容，所有的5个replica都加到一个新的ReplicaSet中。如果使用比例扩容，新添加的replica将传播到所有的ReplicaSet中。大的部分加入replica数最多的ReplicaSet中，小的部分加入到replica数少的ReplciaSet中。0个replica的ReplicaSet不会被扩容。

在我们上面的例子中，3个replica将添加到旧的ReplicaSet中，2个replica将添加到新的ReplicaSet中。rollout进程最终会将所有的replica移动到新的ReplicaSet中，假设新的replica成为健康状态。

```
$ kubectl get deploy
NAME                  DESIRED   CURRENT   UP-TO-DATE   AVAILABLE
  AGE
nginx-deployment      15        18        7            8
  7m
$ kubectl get rs
NAME                           DESIRED   CURRENT   READY     AGE
nginx-deployment-1989198191    7         7         0         7m
nginx-deployment-618515232     11        11        11        7m
```

暂停和恢复Deployment

您可以在发出一次或多次更新前暂停一个 Deployment，然后再恢复它。这样您就能多次暂停和恢复 Deployment，在此期间进行一些修复工作，而不会发出不必要的 rollout。

例如使用刚刚创建 Deployment：

```
$ kubectl get deploy
NAME      DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
nginx     3         3         3            3           1m
[mkargaki@dhcp129-211 kubernetes]$ kubectl get rs
NAME               DESIRED   CURRENT   READY     AGE
nginx-2142116321   3         3         3         1m
```

使用以下命令暂停 Deployment：

```
$ kubectl rollout pause deployment/nginx-deployment
deployment "nginx-deployment" paused
```

然后更新 Deplyment中的镜像：

```
$ kubectl set image deploy/nginx nginx=nginx:1.9.1
deployment "nginx-deployment" image updated
```

注意新的 rollout 启动了：

```
$ kubectl rollout history deploy/nginx
deployments "nginx"
REVISION  CHANGE-CAUSE
1   <none>

$ kubectl get rs
NAME               DESIRED   CURRENT   READY     AGE
nginx-2142116321   3         3         3         2m
```

您可以进行任意多次更新，例如更新使用的资源：

```
$ kubectl set resources deployment nginx -c=nginx --limits=cpu=2
00m,memory=512Mi
deployment "nginx" resource requirements updated
```

Deployment 暂停前的初始状态将继续它的功能，而不会对 Deployment 的更新产生任何影响，只要 Deployment是暂停的。

最后，恢复这个 Deployment，观察完成更新的 ReplicaSet 已经创建出来了：

```
$ kubectl rollout resume deploy nginx
deployment "nginx" resumed
$ KUBECTL get rs -w
NAME               DESIRED   CURRENT   READY     AGE
nginx-2142116321   2         2         2         2m
nginx-3926361531   2         2         0         6s
nginx-3926361531   2         2         1         18s
nginx-2142116321   1         2         2         2m
nginx-2142116321   1         2         2         2m
nginx-3926361531   3         2         1         18s
nginx-3926361531   3         2         1         18s
nginx-2142116321   1         1         1         2m
nginx-3926361531   3         3         1         18s
nginx-3926361531   3         3         2         19s
nginx-2142116321   0         1         1         2m
nginx-2142116321   0         1         1         2m
nginx-2142116321   0         0         0         2m
nginx-3926361531   3         3         3         20s
^C
$ KUBECTL get rs
NAME               DESIRED   CURRENT   READY     AGE
nginx-2142116321   0         0         0         2m
nginx-3926361531   3         3         3         28s
```

注意： 在恢复 Deployment 之前您无法回退一个已经暂停的 Deployment。

Deployment 状态

Deployment 在生命周期中有多种状态。在创建一个新的 ReplicaSet 的时候它可以是 progressing 状态， complete 状态，或者 fail to progress 状态。

进行中的 Deployment

Kubernetes 将执行过下列任务之一的 Deployment 标记为 *progressing* 状态：

- Deployment 正在创建新的ReplicaSet过程中。
- Deployment 正在扩容一个已有的 ReplicaSet。
- Deployment 正在缩容一个已有的 ReplicaSet。
- 有新的可用的 pod 出现。

您可以使用 `kubectl rollout status` 命令监控 Deployment 的进度。

完成的 Deployment

Kubernetes 将包括以下特性的 Deployment 标记为 *complete* 状态：

- Deployment 最小可用。最小可用意味着 Deployment 的可用 replica 个数等于或者超过 Deployment 策略中的期望个数。
- 所有与该 Deployment 相关的replica都被更新到了您指定版本，也就说更新完成。
- 该 Deployment 中没有旧的 Pod 存在。

您可以用 `kubectl rollout status` 命令查看 Deployment 是否完成。如果 rollout 成功完成， `kubectl rollout status` 将返回一个0值的 Exit Code。

```
$ kubectl rollout status deploy/nginx
Waiting for rollout to finish: 2 of 3 updated replicas are avail
able...
deployment "nginx" successfully rolled out
$ echo $?
0
```

失败的 Deployment

您的 Deployment 在尝试部署新的 ReplicaSet 的时候可能卡住，永远也不会完成。这可能是因为以下几个因素引起的：

- 无效的引用
- 不可读的 probe failure
- 镜像拉取错误
- 权限不够
- 范围限制
- 程序运行时配置错误

探测这种情况的一种方式是，在您的 Deployment spec 中指定 `spec.progressDeadlineSeconds`。`spec.progressDeadlineSeconds` 表示 Deployment controller 等待多少秒才能确定（通过 Deployment status）Deployment进程是卡住的。

下面的 `kubectl` 命令设置 `progressDeadlineSeconds` 使 controller 在 Deployment 在进度卡住10分钟后报告：

```
$ kubectl patch deployment/nginx-deployment -p '{"spec":{"progressDeadlineSeconds":600}}'
"nginx-deployment" patched
```

当超过截止时间后，Deployment controller 会在 Deployment 的 `status.conditions` 中增加一条DeploymentCondition，它包括如下属性：

- Type=Progressing
- Status=False
- Reason=ProgressDeadlineExceeded

浏览 Kubernetes API conventions 查看关于status conditions的更多信息。

注意：kubernetes除了报告 `Reason=ProgressDeadlineExceeded` 状态信息外不会对卡住的 Deployment 做任何操作。更高层次的协调器可以利用它并采取相应行动，例如，回滚 Deployment 到之前的版本。

注意：如果您暂停了一个 Deployment，在暂停的这段时间内kubernetnes不会检查您指定的 deadline。您可以在 Deployment 的 rollout 途中安全的暂停它，然后再恢复它，这不会触发超过deadline的状态。

您可能在使用 Deployment 的时候遇到一些短暂的错误，这些可能是由于您设置了太短的 timeout，也有可能是因为各种其他错误导致的短暂错误。例如，假设您使用了无效的引用。当您 Describe Deployment 的时候可能会注意到如下信息：

```
$ kubectl describe deployment nginx-deployment
<...>
Conditions:
  Type            Status  Reason
  ----            ------  ------
  Available       True    MinimumReplicasAvailable
  Progressing     True    ReplicaSetUpdated
  ReplicaFailure  True    FailedCreate
<...>
```

执行 `kubectl get deployment nginx-deployment -o yaml`，Deployement 的状态可能看起来像这个样子：

```
status:
  availableReplicas: 2
  conditions:
  - lastTransitionTime: 2016-10-04T12:25:39Z
    lastUpdateTime: 2016-10-04T12:25:39Z
    message: Replica set "nginx-deployment-4262182780" is progre
ssing.
    reason: ReplicaSetUpdated
    status: "True"
    type: Progressing
  - lastTransitionTime: 2016-10-04T12:25:42Z
    lastUpdateTime: 2016-10-04T12:25:42Z
    message: Deployment has minimum availability.
    reason: MinimumReplicasAvailable
    status: "True"
    type: Available
  - lastTransitionTime: 2016-10-04T12:25:39Z
    lastUpdateTime: 2016-10-04T12:25:39Z
    message: 'Error creating: pods "nginx-deployment-4262182780-
" is forbidden: exceeded quota:
      object-counts, requested: pods=1, used: pods=3, limited: p
ods=2'
    reason: FailedCreate
    status: "True"
    type: ReplicaFailure
  observedGeneration: 3
  replicas: 2
  unavailableReplicas: 2
```

最终，一旦超过 Deployment 进程的 deadline，kuberentes 会更新状态和导致 Progressing 状态的原因：

```
Conditions:
  Type            Status  Reason
  ----            ------  ------
  Available       True    MinimumReplicasAvailable
  Progressing     False   ProgressDeadlineExceeded
  ReplicaFailure  True    FailedCreate
```

您可以通过缩容 Deployment的方式解决配额不足的问题，或者增加您的 namespace 的配额。如果您满足了配额条件后，Deployment controller 就会完成您的 Deployment rollout，您将看到 Deployment 的状态更新为成功状态（`Status=True` 并且 `Reason=NewReplicaSetAvailable`）。

```
Conditions:
  Type          Status  Reason
  ----          ------  ------
  Available     True    MinimumReplicasAvailable
  Progressing   True    NewReplicaSetAvailable
```

`Type=Available` 、 `Status=True` 以为这您的Deployment有最小可用性。 最小可用性是在Deployment策略中指定的参数。 `Type=Progressing` 、 `Status=True` 意味着您的Deployment 或者在部署过程中，或者已经成功部署，达到了期望的最少的可用replica数量（查看特定状态的Reason——在我们的例子中 `Reason=NewReplicaSetAvailable` 意味着Deployment已经完成）。

您可以使用 `kubectl rollout status` 命令查看Deployment进程是否失败。当Deployment过程超过了deadline， `kubectl rollout status` 将返回非0的exit code。

```
$ kubectl rollout status deploy/nginx
Waiting for rollout to finish: 2 out of 3 new replicas have been
 updated...
error: deployment "nginx" exceeded its progress deadline
$ echo $?
1
```

操作失败的 Deployment

所有对完成的 Deployment 的操作都适用于失败的 Deployment。您可以对它扩/缩容，回退到历史版本，您甚至可以多次暂停它来应用 Deployment pod template。

清理Policy

您可以设置 Deployment 中的 `.spec.revisionHistoryLimit` 项来指定保留多少旧的 ReplicaSet。 余下的将在后台被当作垃圾收集。默认的，所有的 revision 历史就都会被保留。在未来的版本中，将会更改为2。

注意： 将该值设置为0，将导致所有的 Deployment 历史记录都会被清除，该 Deployment 就无法再回退了。

用例

金丝雀 Deployment

如果您想要使用 Deployment 对部分用户或服务器发布 release，您可以创建多个 Deployment，每个 Deployment 对应一个 release，参照 managing resources 中对金丝雀模式的描述。

编写 Deployment Spec

在所有的 Kubernetes 配置中，Deployment 也需要 `apiVersion`，`kind` 和 `metadata` 这些配置项。配置文件的通用使用说明查看 部署应用，配置容器，和 使用 kubectl 管理资源 文档。

Deployment也需要 `.spec` section.

Pod Template

`.spec.template` 是 `.spec` 中唯一要求的字段。

`.spec.template` 是 pod template. 它跟 Pod有一模一样的schema，除了它是嵌套的并且不需要 `apiVersion` 和 `kind` 字段。

另外为了划分Pod的范围，Deployment中的pod template必须指定适当的label（不要跟其他controller重复了，参考selector）和适当的重启策略。

`.spec.template.spec.restartPolicy` 可以设置为 `Always` ，如果不指定的话这就是默认配置。

Replicas

`.spec.replicas` 是可以选字段，指定期望的pod数量，默认是1。

Selector

`.spec.selector` 是可选字段，用来指定 label selector ，圈定Deployment管理的pod范围。

如果被指定， `.spec.selector` 必须匹配 `.spec.template.metadata.labels` ，否则它将被API拒绝。如果 `.spec.selector` 没有被指定， `.spec.selector.matchLabels` 默认是 `.spec.template.metadata.labels` 。

在Pod的template跟 `.spec.template` 不同或者数量超过了 `.spec.replicas` 规定的数量的情况下，Deployment会杀掉label跟selector不同的Pod。

注意： 您不应该再创建其他label跟这个selector匹配的pod，或者通过其他Deployment，或者通过其他Controller，例如ReplicaSet和ReplicationController。否则该Deployment会被把它们当成都是自己创建的。Kubernetes不会阻止您这么做。

如果您有多个controller使用了重复的selector，controller们就会互相打架并导致不正确的行为。

策略

`.spec.strategy` 指定新的Pod替换旧的Pod的策略。 `.spec.strategy.type` 可以是"Recreate"或者是 "RollingUpdate"。"RollingUpdate"是默认值。

Recreate Deployment

`.spec.strategy.type==Recreate` 时，在创建出新的Pod之前会先杀掉所有已存在的Pod。

Rolling Update Deployment

`.spec.strategy.type==RollingUpdate` 时，Deployment使用rolling update 的方式更新Pod 。您可以指定 `maxUnavailable` 和 `maxSurge` 来控制 rolling update 进程。

Max Unavailable

`.spec.strategy.rollingUpdate.maxUnavailable` 是可选配置项，用来指定在升级过程中不可用Pod的最大数量。该值可以是一个绝对值（例如5），也可以是期望Pod数量的百分比（例如10%）。通过计算百分比的绝对值向下取整。如果 `.spec.strategy.rollingUpdate.maxSurge` 为0时，这个值不可以为0。默认值是1。

例如，该值设置成30%，启动rolling update后旧的ReplicatSet将会立即缩容到期望的Pod数量的70%。新的Pod ready后，随着新的ReplicaSet的扩容，旧的ReplicaSet会进一步缩容，确保在升级的所有时刻可以用的Pod数量至少是期望Pod数量的70%。

Max Surge

`.spec.strategy.rollingUpdate.maxSurge` 是可选配置项，用来指定可以超过期望的Pod数量的最大个数。该值可以是一个绝对值（例如5）或者是期望的Pod数量的百分比（例如10%）。当 `MaxUnavailable` 为0时该值不可以为0。通过百分比计算的绝对值向上取整。默认值是1。

例如，该值设置成30%，启动rolling update后新的ReplicatSet将会立即扩容，新老Pod的总数不能超过期望的Pod数量的130%。旧的Pod被杀掉后，新的ReplicaSet将继续扩容，旧的ReplicaSet会进一步缩容，确保在升级的所有时刻所有的Pod数量和不会超过期望Pod数量的130%。

Progress Deadline Seconds

`.spec.progressDeadlineSeconds` 是可选配置项，用来指定在系统报告Deployment的failed progressing ——表现为resource的状态中 `type=Progressing` 、 `Status=False` 、

`Reason=ProgressDeadlineExceeded` 前可以等待的Deployment进行的秒数。Deployment controller会继续重试该Deployment。未来，在实现了自动回滚后，deployment controller在观察到这种状态时就会自动回滚。

如果设置该参数，该值必须大于 `.spec.minReadySeconds` 。

Min Ready Seconds

`.spec.minReadySeconds` 是一个可选配置项，用来指定没有任何容器crash的Pod并被认为是可用状态的最小秒数。默认是0（Pod在ready后就会被认为是可用状态）。进一步了解什么什么后Pod会被认为是ready状态，参阅 Container Probes。

Rollback To

`.spec.rollbackTo` 是一个可以选配置项，用来配置Deployment回退的配置。设置该参数将触发回退操作，每次回退完成后，该值就会被清除。

Revision

`.spec.rollbackTo.revision` 是一个可选配置项，用来指定回退到的revision。默认是0，意味着回退到上一个revision。

Revision History Limit

Deployment revision history存储在它控制的ReplicaSets中。

`.spec.revisionHistoryLimit` 是一个可选配置项，用来指定可以保留的旧的ReplicaSet数量。该理想值取决于心Deployment的频率和稳定性。如果该值没有设置的话，默认所有旧的Replicaset或会被保留，将资源存储在etcd中，是用 `kubectl get rs` 查看输出。每个Deployment的该配置都保存在ReplicaSet中，然而，一旦您删除的旧的RepelicaSet，您的Deployment就无法再回退到那个revison了。

如果您将该值设置为0，所有具有0个replica的ReplicaSet都会被删除。在这种情况下，新的Deployment rollout无法撤销，因为revision history都被清理掉了。

Paused

`.spec.paused` 是可以可选配置项，boolean值。用来指定暂停和恢复Deployment。Paused和没有paused的Deployment之间的唯一区别就是，所有对paused deployment中的PodTemplateSpec的修改都不会触发新的rollout。Deployment被创建之后默认是非paused。

Deployment 的替代选择

kubectl rolling update

Kubectl rolling update 虽然使用类似的方式更新Pod和ReplicationController。但是我们推荐使用Deployment，因为它是声明式的，客户端侧，具有附加特性，例如即使滚动升级结束后也可以回滚到任何历史版本。

StatefulSet

StatefulSet 作为 Controller 为 Pod 提供唯一的标识。它可以保证部署和 scale 的顺序。

使用案例参考：kubernetes contrib - statefulsets，其中包含zookeeper和kakfa的statefulset设置和使用说明。

StatefulSet是为了解决有状态服务的问题（对应Deployments和ReplicaSets是为无状态服务而设计），其应用场景包括：

- 稳定的持久化存储，即Pod重新调度后还是能访问到相同的持久化数据，基于PVC来实现
- 稳定的网络标志，即Pod重新调度后其PodName和HostName不变，基于Headless Service（即没有Cluster IP的Service）来实现
- 有序部署，有序扩展，即Pod是有顺序的，在部署或者扩展的时候要依据定义的顺序依次依次进行（即从0到N-1，在下一个Pod运行之前所有之前的Pod必须都是Running和Ready状态），基于init containers来实现
- 有序收缩，有序删除（即从N-1到0）

从上面的应用场景可以发现，StatefulSet由以下几个部分组成：

- 用于定义网络标志（DNS domain）的Headless Service
- 用于创建PersistentVolumes的volumeClaimTemplates
- 定义具体应用的StatefulSet

StatefulSet中每个Pod的DNS格式为 `statefulSetName-{0..N-1}.serviceName.namespace.svc.cluster.local` ，其中

- `serviceName` 为Headless Service的名字
- `0..N-1` 为Pod所在的序号，从0开始到N-1
- `statefulSetName` 为StatefulSet的名字
- `namespace` 为服务所在的namespace，Headless Servic和StatefulSet必须在相同的namespace
- `.cluster.local` 为Cluster Domain

使用 StatefulSet

StatefulSet 适用于有以下某个或多个需求的应用：

- 稳定，唯一的网络标志。
- 稳定，持久化存储。
- 有序，优雅地部署和 scale。
- 有序，优雅地删除和终止。
- 有序，自动的滚动升级。

在上文中，稳定是 Pod （重新）调度中持久性的代名词。 如果应用程序不需要任何稳定的标识符、有序部署、删除和 scale，则应该使用提供一组无状态副本的 controller 来部署应用程序，例如 Deployment 或 ReplicaSet 可能更适合您的无状态需求。

限制

- StatefulSet 是 beta 资源，Kubernetes 1.5 以前版本不支持。
- 对于所有的 alpha/beta 的资源，您都可以通过在 apiserver 中设置 `--runtime-config` 选项来禁用。
- 给定 Pod 的存储必须由 PersistentVolume Provisioner 根据请求的 `storage class` 进行配置，或由管理员预先配置。
- 删除或 scale StatefulSet 将不会删除与 StatefulSet 相关联的 volume。 这样做是为了确保数据安全性，这通常比自动清除所有相关 StatefulSet 资源更有价值。
- StatefulSets 目前要求 Headless Service 负责 Pod 的网络身份。 您有责任创建此服务。

组件

下面的示例中描述了 StatefulSet 中的组件。

- 一个名为 nginx 的 headless service，用于控制网络域。
- 一个名为 web 的 StatefulSet，它的 Spec 中指定在有 3 个运行 nginx 容器的 Pod。
- volumeClaimTemplates 使用 PersistentVolume Provisioner 提供的 PersistentVolumes 作为稳定存储。

```yaml
apiVersion: v1
kind: Service
metadata:
  name: nginx
  labels:
    app: nginx
spec:
  ports:
  - port: 80
    name: web
  clusterIP: None
  selector:
    app: nginx
---
apiVersion: apps/v1beta1
kind: StatefulSet
metadata:
  name: web
spec:
  serviceName: "nginx"
  replicas: 3
  template:
    metadata:
      labels:
        app: nginx
    spec:
      terminationGracePeriodSeconds: 10
      containers:
      - name: nginx
        image: gcr.io/google_containers/nginx-slim:0.8
        ports:
        - containerPort: 80
          name: web
        volumeMounts:
        - name: www
          mountPath: /usr/share/nginx/html
  volumeClaimTemplates:
  - metadata:
      name: www
      annotations:
        volume.beta.kubernetes.io/storage-class: anything
    spec:
      accessModes: [ "ReadWriteOnce" ]
      resources:
        requests:
          storage: 1Gi
```

Pod 身份

StatefulSet Pod 具有唯一的身份，包括序数，稳定的网络身份和稳定的存储。 身份绑定到 Pod 上，不管它（重新）调度到哪个节点上。

序数

对于一个有 N 个副本的 StatefulSet，每个副本都会被指定一个整数序数，在 [0,N) 之间，且唯一。

稳定的网络 ID

StatefulSet 中的每个 Pod 从 StatefulSet 的名称和 Pod 的序数派生其主机名。构造的主机名的模式是 `$（statefulset名称)-$(序数)` 。 上面的例子将创建三个名为 `web-0，web-1，web-2` 的 Pod。

StatefulSet 可以使用 Headless Service 来控制其 Pod 的域。此服务管理的域的格式为： `$(服务名称).$(namespace).svc.cluster.local` ，其中 “cluster.local” 是集群域。

在创建每个Pod时，它将获取一个匹配的 DNS 子域，采用以下形式： `$(pod 名称).$(管理服务域)` ，其中管理服务由 StatefulSet 上的 `serviceName` 字段定义。

以下是 Cluster Domain，服务名称，StatefulSet 名称以及如何影响 StatefulSet 的 Pod 的 DNS 名称的一些示例。

Cluster Domain	Service (ns/name)	StatefulSet (ns/name)	StatefulSet Domain	
cluster.local	default/nginx	default/web	nginx.default.svc.cluster.local	w 1
cluster.local	foo/nginx	foo/web	nginx.foo.svc.cluster.local	w 1
kube.local	foo/nginx	foo/web	nginx.foo.svc.kube.local	w 1

注意 Cluster Domain 将被设置成 `cluster.local` 除非进行了 其他配置。

稳定存储

Kubernetes 为每个 VolumeClaimTemplate 创建一个 PersistentVolume。上面的 nginx 的例子中，每个 Pod 将具有一个由 `anything` 存储类创建的 1 GB 存储的 PersistentVolume。当该 Pod （重新）调度到节点上，`volumeMounts` 将挂载与 PersistentVolume Claim 相关联的 PersistentVolume。请注意，与 PersistentVolume Claim 相关联的 PersistentVolume 在 产出 Pod 或 StatefulSet 的时候不会被删除。这必须手动完成。

部署和 Scale 保证

- 对于有 N 个副本的 StatefulSet，Pod 将按照 {0..N-1} 的顺序被创建和部署。
- 当 删除 Pod 的时候，将按照逆序来终结，从{N-1..0}
- 对 Pod 执行 scale 操作之前，它所有的前任必须处于 Running 和 Ready 状态。
- 在终止 Pod 前，它所有的继任者必须处于完全关闭状态。

不应该将 StatefulSet 的 `pod.Spec.TerminationGracePeriodSeconds` 设置为 0。这样是不安全的且强烈不建议您这样做。进一步解释，请参阅 强制删除 StatefulSet Pod。

上面的 nginx 示例创建后，3 个 Pod 将按照如下顺序创建 web-0，web-1，web-2。在 web-0 处于 运行并就绪 状态之前，web-1 将不会被部署，同样当 web-1 处于运行并就绪状态之前 web-2也不会被部署。如果在 web-1 运行并就绪后，web-2 启动之前， web-0 失败了，web-2 将不会启动，直到 web-0 成功重启并处于运行并就绪状态。

如果用户通过修补 StatefulSet 来 scale 部署的示例，以使 `replicas=1` ，则 web-2 将首先被终止。 在 web-2 完全关闭和删除之前，web-1 不会被终止。 如果 web-0 在 web-2 终止并且完全关闭之后，但是在 web-1 终止之前失败，则 web-1 将不会终止，除非 web-0 正在运行并准备就绪。

Pod 管理策略

在 Kubernetes 1.7 和之后版本，StatefulSet 允许您放开顺序保证，同时通过 `.spec.podManagementPolicy` 字段保证身份的唯一性。

OrderedReady Pod 管理

StatefulSet 中默认使用的是 `OrderedReady` pod 管理。它实现了 如上 所述的行为。

并行 Pod 管理

`Parallel` pod 管理告诉 StatefulSet controller 并行的启动和终止 Pod，在启动和终止其他 Pod 之前不会等待 Pod 变成 运行并就绪或完全终止状态。

更新策略

在 kubernetes 1.7 和以上版本中，StatefulSet 的 `.spec.updateStrategy` 字段允许您配置和禁用 StatefulSet 中的容器、label、resource request/limit、annotation 的滚动更新。

删除

`OnDelete` 更新策略实现了遗留（1.6和以前）的行为。 当 `spec.updateStrategy` 未指定时，这是默认策略。 当StatefulSet 的 `.spec.updateStrategy.type` 设置为 `OnDelete` 时，StatefulSet 控制器将不会自动更新 `StatefulSet` 中的 Pod。 用户必须手动删除 Pod 以使控制器创建新的 Pod，以反映对StatefulSet的 `.spec.template` 进行的修改。

滚动更新

`RollingUpdate` 更新策略在 StatefulSet 中实现 Pod 的自动滚动更新。 当StatefulSet的 `.spec.updateStrategy.type` 设置为 `RollingUpdate` 时，StatefulSet 控制器将在 StatefulSet 中删除并重新创建每个 Pod。 它将以与 Pod 终止相同的顺序进行（从最大的序数到最小的序数），每次更新一个 Pod。 在更新其前身之前，它将等待正在更新的 Pod 状态变成正在运行并就绪。

分区

可以通过指定 `.spec.updateStrategy.rollingUpdate.partition` 来对 `RollingUpdate` 更新策略进行分区。如果指定了分区，则当 StatefulSet 的 `.spec.template` 更新时，具有大于或等于分区序数的所有 Pod 将被更新。具有

小于分区的序数的所有 Pod 将不会被更新，即使删除它们也将被重新创建。如果 StatefulSet 的 `.spec.updateStrategy.rollingUpdate.partition` 大于其 `.spec.replicas`，则其 `.spec.template` 的更新将不会传播到 Pod。

在大多数情况下，您不需要使用分区，但如果您想要进行分阶段更新，使用金丝雀发布或执行分阶段发布，它们将非常有用。

简单示例

以一个简单的nginx服务web.yaml为例：

```yaml
---
apiVersion: v1
kind: Service
metadata:
  name: nginx
  labels:
    app: nginx
spec:
  ports:
  - port: 80
    name: web
  clusterIP: None
  selector:
    app: nginx
---
apiVersion: apps/v1beta1
kind: StatefulSet
metadata:
  name: web
spec:
  serviceName: "nginx"
  replicas: 2
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: gcr.io/google_containers/nginx-slim:0.8
        ports:
        - containerPort: 80
          name: web
        volumeMounts:
        - name: www
          mountPath: /usr/share/nginx/html
  volumeClaimTemplates:
  - metadata:
      name: www
      annotations:
        volume.alpha.kubernetes.io/storage-class: anything
    spec:
      accessModes: [ "ReadWriteOnce" ]
      resources:
        requests:
          storage: 1Gi
```

```
$ kubectl create -f web.yaml
service "nginx" created
statefulset "web" created

# 查看创建的headless service和statefulset
$ kubectl get service nginx
NAME      CLUSTER-IP   EXTERNAL-IP   PORT(S)   AGE
nginx     None         <none>        80/TCP    1m
$ kubectl get statefulset web
NAME      DESIRED   CURRENT   AGE
web       2         2         2m

# 根据volumeClaimTemplates自动创建PVC（在GCE中会自动创建kubernetes.io
/gce-pd类型的volume）
$ kubectl get pvc
NAME        STATUS    VOLUME                                     
 CAPACITY   ACCESSMODES   AGE
www-web-0   Bound     pvc-d064a004-d8d4-11e6-b521-42010a800002
 1Gi        RWO           16s
www-web-1   Bound     pvc-d06a3946-d8d4-11e6-b521-42010a800002
 1Gi        RWO           16s

# 查看创建的Pod，他们都是有序的
$ kubectl get pods -l app=nginx
NAME      READY     STATUS    RESTARTS   AGE
web-0     1/1       Running   0          5m
web-1     1/1       Running   0          4m

# 使用nslookup查看这些Pod的DNS
$ kubectl run -i --tty --image busybox dns-test --restart=Never
--rm /bin/sh
/ # nslookup web-0.nginx
Server:    10.0.0.10
Address 1: 10.0.0.10 kube-dns.kube-system.svc.cluster.local

Name:      web-0.nginx
Address 1: 10.244.2.10
/ # nslookup web-1.nginx
Server:    10.0.0.10
Address 1: 10.0.0.10 kube-dns.kube-system.svc.cluster.local

Name:      web-1.nginx
Address 1: 10.244.3.12
/ # nslookup web-0.nginx.default.svc.cluster.local
Server:    10.0.0.10
Address 1: 10.0.0.10 kube-dns.kube-system.svc.cluster.local

Name:      web-0.nginx.default.svc.cluster.local
Address 1: 10.244.2.10
```

还可以进行其他的操作

```
# 扩容
$ kubectl scale statefulset web --replicas=5

# 缩容
$ kubectl patch statefulset web -p '{"spec":{"replicas":3}}'

# 镜像更新（目前还不支持直接更新image，需要patch来间接实现）
$ kubectl patch statefulset web --type='json' -p='[{"op": "repla
ce", "path": "/spec/template/spec/containers/0/image", "value":"
gcr.io/google_containers/nginx-slim:0.7"}]'

# 删除StatefulSet和Headless Service
$ kubectl delete statefulset web
$ kubectl delete service nginx

# StatefulSet删除后PVC还会保留着，数据不再使用的话也需要删除
$ kubectl delete pvc www-web-0 www-web-1
```

zookeeper

另外一个更能说明StatefulSet强大功能的示例为zookeeper.yaml，这个例子仅为讲解，实际可用的配置请使用 https://github.com/kubernetes/contrib/tree/master/statefulsets 中的配置。

```
---
apiVersion: v1
kind: Service
metadata:
  name: zk-headless
  labels:
    app: zk-headless
spec:
  ports:
  - port: 2888
    name: server
  - port: 3888
    name: leader-election
  clusterIP: None
  selector:
    app: zk
---
apiVersion: v1
kind: ConfigMap
metadata:
  name: zk-config
data:
  ensemble: "zk-0;zk-1;zk-2"
```

```
    jvm.heap: "2G"
    tick: "2000"
    init: "10"
    sync: "5"
    client.cnxns: "60"
    snap.retain: "3"
    purge.interval: "1"
---
apiVersion: policy/v1beta1
kind: PodDisruptionBudget
metadata:
  name: zk-budget
spec:
  selector:
    matchLabels:
      app: zk
  minAvailable: 2
---
apiVersion: apps/v1beta1
kind: StatefulSet
metadata:
  name: zk
spec:
  serviceName: zk-headless
  replicas: 3
  template:
    metadata:
      labels:
        app: zk
      annotations:
        pod.alpha.kubernetes.io/initialized: "true"
        scheduler.alpha.kubernetes.io/affinity: >
            {
              "podAntiAffinity": {
                "requiredDuringSchedulingRequiredDuringExecution
": [{
                  "labelSelector": {
                    "matchExpressions": [{
                      "key": "app",
                      "operator": "In",
                      "values": ["zk-headless"]
                    }]
                  },
                  "topologyKey": "kubernetes.io/hostname"
                }]
              }
            }
    spec:
      containers:
      - name: k8szk
        imagePullPolicy: Always
        image: gcr.io/google_samples/k8szk:v1
        resources:
```

```
          requests:
            memory: "4Gi"
            cpu: "1"
        ports:
        - containerPort: 2181
          name: client
        - containerPort: 2888
          name: server
        - containerPort: 3888
          name: leader-election
        env:
        - name : ZK_ENSEMBLE
          valueFrom:
            configMapKeyRef:
              name: zk-config
              key: ensemble
        - name : ZK_HEAP_SIZE
          valueFrom:
            configMapKeyRef:
                name: zk-config
                key: jvm.heap
        - name : ZK_TICK_TIME
          valueFrom:
            configMapKeyRef:
                name: zk-config
                key: tick
        - name : ZK_INIT_LIMIT
          valueFrom:
            configMapKeyRef:
                name: zk-config
                key: init
        - name : ZK_SYNC_LIMIT
          valueFrom:
            configMapKeyRef:
                name: zk-config
                key: tick
        - name : ZK_MAX_CLIENT_CNXNS
          valueFrom:
            configMapKeyRef:
                name: zk-config
                key: client.cnxns
        - name: ZK_SNAP_RETAIN_COUNT
          valueFrom:
            configMapKeyRef:
                name: zk-config
                key: snap.retain
        - name: ZK_PURGE_INTERVAL
          valueFrom:
            configMapKeyRef:
                name: zk-config
                key: purge.interval
        - name: ZK_CLIENT_PORT
          value: "2181"
```

```yaml
        - name: ZK_SERVER_PORT
          value: "2888"
        - name: ZK_ELECTION_PORT
          value: "3888"
        command:
        - sh
        - -c
        - zkGenConfig.sh && zkServer.sh start-foreground
        readinessProbe:
          exec:
            command:
            - "zkOk.sh"
          initialDelaySeconds: 15
          timeoutSeconds: 5
        livenessProbe:
          exec:
            command:
            - "zkOk.sh"
          initialDelaySeconds: 15
          timeoutSeconds: 5
        volumeMounts:
        - name: datadir
          mountPath: /var/lib/zookeeper
      securityContext:
        runAsUser: 1000
        fsGroup: 1000
  volumeClaimTemplates:
  - metadata:
      name: datadir
      annotations:
        volume.alpha.kubernetes.io/storage-class: anything
    spec:
      accessModes: [ "ReadWriteOnce" ]
      resources:
        requests:
          storage: 20Gi
```

```
kubectl create -f zookeeper.yaml
```

详细的使用说明见zookeeper stateful application。

关于StatefulSet的更多示例请参阅 github.com/kubernetes/contrib - statefulsets，其中包括了zookeeper和kafka。

集群外部访问StatefulSet的Pod

我们设想一下这样的场景：在kubernetes集群外部调试StatefulSet中有序的Pod，那么如何访问这些的pod呢？

方法是为pod设置label，然后用 `kubectl expose` 将其以NodePort的方式暴露到集群外部，以上面的zookeeper的例子来说明，下面使用命令的方式来暴露其中的两个zookeeper节点，也可以写一个serivce配置yaml文件。

```
kubectl label pod zk-0 zkInst=0

kubectl label pod zk-1 zkInst=1

kubectl expose po zk-0 --port=2181 --target-port=2181 --name=zk-0 --selector=zkInst=0 --type=NodePort
kubectl expose po zk-1 --port=2181 --target-port=2181 --name=zk-1 --selector=zkInst=1 --type=NodePort
```

这样在kubernetes集群外部就可以根据pod所在的主机所映射的端口来访问了。

查看 `zk-0` 这个service可以看到如下结果：

```
NAME      CLUSTER-IP      EXTERNAL-IP   PORT(S)          AGE
zk-0      10.254.98.14    <nodes>       2181:31693/TCP   5m
```

集群外部就可以使用所有的node中的任何一个IP:31693来访问这个zookeeper实例。

参考

https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/

kubernetes contrib - statefulsets

DaemonSet

什么是 DaemonSet？

DaemonSet 确保全部（或者一些）Node 上运行一个 Pod 的副本。当有 Node 加入集群时，也会为他们新增一个 Pod 。当有 Node 从集群移除时，这些 Pod 也会被回收。删除 DaemonSet 将会删除它创建的所有 Pod。

使用 DaemonSet 的一些典型用法：

- 运行集群存储 daemon，例如在每个 Node 上运行 `glusterd` 、 `ceph` 。
- 在每个 Node 上运行日志收集 daemon，例如 `fluentd` 、 `logstash` 。
- 在每个 Node 上运行监控 daemon，例如 Prometheus Node Exporter、 `collectd` 、Datadog 代理、New Relic 代理，或 Ganglia `gmond` 。

一个简单的用法是，在所有的 Node 上都存在一个 DaemonSet，将被作为每种类型的 daemon 使用。 一个稍微复杂的用法可能是，对单独的每种类型的 daemon 使用多个 DaemonSet，但具有不同的标志，和/或对不同硬件类型具有不同的内存、CPU要求。

编写 DaemonSet Spec

必需字段

和其它所有 Kubernetes 配置一样，DaemonSet 需要 `apiVersion` 、 `kind` 和 `metadata` 字段。有关配置文件的通用信息，详见文档 部署应用 、配置容器 和 资源管理 。

DaemonSet 也需要一个 `.spec` 配置段。

Pod 模板

`.spec` 唯一必需的字段是 `.spec.template` 。

`.spec.template` 是一个 Pod 模板。 它与 Pod 具有相同的 schema，除了它是嵌套的，而且不具有 `apiVersion` 或 `kind` 字段。

Pod 除了必须字段外，在 DaemonSet 中的 Pod 模板必须指定合理的标签（查看 pod selector）。

在 DaemonSet 中的 Pod 模板必需具有一个值为 `Always` 的 `RestartPolicy`，或者未指定它的值，默认是 `Always`。

Pod Selector

`.spec.selector` 字段表示 Pod Selector，它与 Job 或其它资源的 `.sper.selector` 的原理是相同的。

`spec.selector` 表示一个对象，它由如下两个字段组成：

- `matchLabels` - 与 ReplicationController 的 `.spec.selector` 的原理相同。
- `matchExpressions` - 允许构建更加复杂的 Selector，可以通过指定 key、value 列表，以及与 key 和 value 列表的相关的操作符。

当上述两个字段都指定时，结果表示的是 AND 关系。

如果指定了 `.spec.selector`，必须与 `.spec.template.metadata.labels` 相匹配。如果没有指定，它们默认是等价的。如果与它们配置的不匹配，则会被 API 拒绝。

如果 Pod 的 label 与 selector 匹配，或者直接基于其它的 DaemonSet、或者 Controller（例如 ReplicationController），也不可以创建任何 Pod。 否则 DaemonSet Controller 将认为那些 Pod 是它创建的。Kubernetes 不会阻止这样做。一个场景是，可能希望在一个具有不同值的、用来测试用的 Node 上手动创建 Pod。

仅在相同的 Node 上运行 Pod

如果指定了 `.spec.template.spec.nodeSelector`，DaemonSet Controller 将在能够匹配上 Node Selector 的 Node 上创建 Pod。 类似这种情况，可以指定 `.spec.template.spec.affinity`，然后 DaemonSet Controller 将在能够匹配

上 Node Affinity 的 Node 上创建 Pod。 如果根本就没有指定，则 DaemonSet Controller 将在所有 Node 上创建 Pod。

如何调度 Daemon Pod

正常情况下，Pod 运行在哪个机器上是由 Kubernetes 调度器进行选择的。然而，由 Daemon Controller 创建的 Pod 已经确定了在哪个机器上（Pod 创建时指定了 `.spec.nodeName` ），因此：

- DaemonSet Controller 并不关心一个 Node 的 `unschedulable` 字段。
- DaemonSet Controller 可以创建 Pod，即使调度器还没有被启动，这对集群启动是非常有帮助的。

Daemon Pod 关心 Taint 和 Toleration，它们会为没有指定 `tolerationSeconds` 的 `node.alpha.kubernetes.io/notReady` 和 `node.alpha.kubernetes.io/unreachable` 的 Taint，而创建具有 `NoExecute` 的 Toleration。这确保了当 alpha 特性的 `TaintBasedEvictions` 被启用，当 Node 出现故障，比如网络分区，这时它们将不会被清除掉（当 `TaintBasedEvictions` 特性没有启用，在这些场景下也不会被清除，但会因为 NodeController 的硬编码行为而被清除，Toleration 是不会的）。

与 Daemon Pod 通信

与 DaemonSet 中的 Pod 进行通信，几种可能的模式如下：

- **Push**：配置 DaemonSet 中的 Pod 向其它 Service 发送更新，例如统计数据库。它们没有客户端。
- **NodeIP 和已知端口**：DaemonSet 中的 Pod 可以使用 `hostPort` ，从而可以通过 Node IP 访问到 Pod。客户端能通过某种方法知道 Node IP 列表，并且基于此也可以知道端口。
- **DNS**：创建具有相同 Pod Selector 的 Headless Service，然后通过使用 `endpoints` 资源或从 DNS 检索到多个 A 记录来发现 DaemonSet。
- **Service**：创建具有相同 Pod Selector 的 Service，并使用该 Service 访问到某个随机 Node 上的 daemon。（没有办法访问到特定 Node）

更新 DaemonSet

如果修改了 Node Label，DaemonSet 将立刻向新匹配上的 Node 添加 Pod，同时删除新近无法匹配上的 Node 上的 Pod。

可以修改 DaemonSet 创建的 Pod。然而，不允许对 Pod 的所有字段进行更新。当下次 Node（即使具有相同的名称）被创建时，DaemonSet Controller 还会使用最初的模板。

可以删除一个 DaemonSet。如果使用 `kubectl` 并指定 `--cascade=false` 选项，则 Pod 将被保留在 Node 上。然后可以创建具有不同模板的新 DaemonSet。具有不同模板的新 DaemonSet 将鞥能够通过 Label 匹配识别所有已经存在的 Pod。它不会修改或删除它们，即使是错误匹配了 Pod 模板。通过删除 Pod 或者删除 Node，可以强制创建新的 Pod。

在 Kubernetes 1.6 或以后版本，可以在 DaemonSet 上 执行滚动升级。

init 脚本

很可能通过直接在一个 Node 上启动 daemon 进程（例如，使用 `init` 、 `upstartd` 、或 `systemd` ）。这非常好，然而基于 DaemonSet 来运行这些进程有如下一些好处：

- 像对待应用程序一样，具备为 daemon 提供监控和管理日志的能力。
- 为 daemon 和应用程序使用相同的配置语言和工具（如 Pod 模板、 `kubectl` ）。
- Kubernetes 未来版本可能会支持对 DaemonSet 创建 Pod 与 Node升级工作流进行集成。
- 在资源受限的容器中运行 daemon，能够增加 daemon 和应用容器的隔离性。然而这也实现了在容器中运行 daemon，但却不能在 Pod 中运行（例如，直接基于 Docker 启动）。

裸 Pod

可能要直接创建 Pod，同时指定其运行在特定的 Node 上。 然而，DaemonSet 替换了由于任何原因被删除或终止的 Pod，例如 Node 失败、例行节点维护，比如内核升级。由于这个原因，我们应该使用 DaemonSet 而不是单独创建 Pod。

静态 Pod

很可能，通过在一个指定目录下编写文件来创建 Pod，该目录受 Kubelet 所监视。这些 Pod 被称为 静态 Pod。 不像 DaemonSet，静态 Pod 不受 kubectl 和 其它 Kubernetes API 客户端管理。静态 Pod 不依赖于 apiserver，这使得它们在集群启动的情况下非常有用。 而且，未来静态 Pod 可能会被废弃掉。

Replication Controller

DaemonSet 与 Replication Controller 非常类似，它们都能创建 Pod，这些 Pod 都具有不期望被终止的进程（例如，Web 服务器、存储服务器）。 为无状态的 Service 使用 Replication Controller，像 frontend，实现对副本的数量进行扩缩容、平滑升级，比之于精确控制 Pod 运行在某个主机上要重要得多。需要 Pod 副本总是运行在全部或特定主机上，并需要先于其他 Pod 启动，当这被认为非常重要时，应该使用 Daemon Controller。

ReplicationController和ReplicaSet

ReplicationCtronller用来确保容器应用的副本数始终保持在用户定义的副本数，即如果有容器异常退出，会自动创建新的Pod来替代；而如果异常多出来的容器也会自动回收。

在新版本的Kubernetes中建议使用ReplicaSet来取代ReplicationCtronller。ReplicaSet跟ReplicationCtronller没有本质的不同，只是名字不一样，并且ReplicaSet支持集合式的selector。

虽然ReplicaSet可以独立使用，但一般还是建议使用 Deployment 来自动管理ReplicaSet，这样就无需担心跟其他机制的不兼容问题（比如ReplicaSet不支持rolling-update但Deployment支持）。

ReplicaSet示例：

```yaml
apiVersion: extensions/v1beta1
kind: ReplicaSet
metadata:
  name: frontend
  # these labels can be applied automatically
  # from the labels in the pod template if not set
  # labels:
    # app: guestbook
    # tier: frontend
spec:
  # this replicas value is default
  # modify it according to your case
  replicas: 3
  # selector can be applied automatically
  # from the labels in the pod template if not set,
  # but we are specifying the selector here to
  # demonstrate its usage.
  selector:
    matchLabels:
      tier: frontend
    matchExpressions:
      - {key: tier, operator: In, values: [frontend]}
  template:
    metadata:
      labels:
        app: guestbook
        tier: frontend
    spec:
      containers:
      - name: php-redis
        image: gcr.io/google_samples/gb-frontend:v3
        resources:
          requests:
            cpu: 100m
            memory: 100Mi
        env:
        - name: GET_HOSTS_FROM
          value: dns
          # If your cluster config does not include a dns servic
e, then to
          # instead access environment variables to find service
 host
          # info, comment out the 'value: dns' line above, and u
ncomment the
          # line below.
          # value: env
        ports:
        - containerPort: 80
```

GitbookUpdated at 2018-05-09 14:15:47

Job

Job负责批处理任务，即仅执行一次的任务，它保证批处理任务的一个或多个Pod成功结束。

Job Spec格式

- spec.template格式同Pod
- RestartPolicy仅支持Never或OnFailure
- 单个Pod时，默认Pod成功运行后Job即结束
- `.spec.completions` 标志Job结束需要成功运行的Pod个数，默认为1
- `.spec.parallelism` 标志并行运行的Pod的个数，默认为1
- `spec.activeDeadlineSeconds` 标志失败Pod的重试最大时间，超过这个时间不会继续重试

一个简单的例子：

```yaml
apiVersion: batch/v1
kind: Job
metadata:
  name: pi
spec:
  template:
    metadata:
      name: pi
    spec:
      containers:
      - name: pi
        image: perl
        command: ["perl",  "-Mbignum=bpi", "-wle", "print bpi(2000)"]
      restartPolicy: Never
```

```
$ kubectl create -f ./job.yaml
job "pi" created
$ pods=$(kubectl get pods --selector=job-name=pi --output=jsonpath={.items..metadata.name})
$ kubectl logs $pods
3.141592653589793238462643383279502...
```

Bare Pods

所谓Bare Pods是指直接用PodSpec来创建的Pod（即不在ReplicaSets或者ReplicationController的管理之下的Pods）。这些Pod在Node重启后不会自动重启，但Job则会创建新的Pod继续任务。所以，推荐使用Job来替代Bare Pods，即便是应用只需要一个Pod。

CronJob

Cron Job 管理基于时间的 Job，即：

- 在给定时间点只运行一次
- 周期性地在给定时间点运行

一个 CronJob 对象类似于 *crontab* （cron table）文件中的一行。它根据指定的预定计划周期性地运行一个 Job，格式可以参考 Cron 。

前提条件

当使用的 Kubernetes 集群，版本 >= 1.4（对 ScheduledJob），>= 1.5（对 CronJob），当启动 API Server（参考 为集群开启或关闭 API 版本 获取更多信息）时，通过传递选项 `--runtime-config=batch/v2alpha1=true` 可以开启 batch/v2alpha1 API。

典型的用法如下所示：

- 在给定的时间点调度 Job 运行
- 创建周期性运行的 Job，例如：数据库备份、发送邮件。

CronJob Spec

- `.spec.schedule` ：调度，必需字段，指定任务运行周期，格式同 Cron
- `.spec.jobTemplate` ：**Job** 模板，必需字段，指定需要运行的任务，格式同 Job
- `.spec.startingDeadlineSeconds` ：启动 **Job** 的期限（秒级别），该字段是可选的。如果因为任何原因而错过了被调度的时间，那么错过执行时间的 Job 将被认为是失败的。如果没有指定，则没有期限
- `.spec.concurrencyPolicy` ：并发策略，该字段也是可选的。它指定了如何处理被 Cron Job 创建的 Job 的并发执行。只允许指定下面策略中的一种：
  - `Allow` （默认）：允许并发运行 Job
  - `Forbid` ：禁止并发运行，如果前一个还没有完成，则直接跳过下一个

    - `Replace`：取消当前正在运行的 Job，用一个新的来替换

  注意，当前策略只能应用于同一个 Cron Job 创建的 Job。如果存在多个 Cron Job，它们创建的 Job 之间总是允许并发运行。

- `.spec.suspend`：挂起，该字段也是可选的。如果设置为 `true`，后续所有执行都会被挂起。它对已经开始执行的 Job 不起作用。默认值为 `false`。

- `.spec.successfulJobsHistoryLimit` 和 `.spec.failedJobsHistoryLimit`：历史限制，是可选的字段。它们指定了可以保留多少完成和失败的 Job。

  默认没有限制，所有成功和失败的 Job 都会被保留。然而，当运行一个 Cron Job 时，Job 可以很快就堆积很多，推荐设置这两个字段的值。设置限制的值为 `0`，相关类型的 Job 完成后将不会被保留。

```yaml
apiVersion: batch/v2alpha1
kind: CronJob
metadata:
  name: hello
spec:
  schedule: "*/1 * * * *"
  jobTemplate:
    spec:
      template:
        spec:
          containers:
          - name: hello
            image: busybox
            args:
            - /bin/sh
            - -c
            - date; echo Hello from the Kubernetes cluster
          restartPolicy: OnFailure
```

```
$ kubectl create -f cronjob.yaml
cronjob "hello" created
```

当然，也可以用 `kubectl run` 来创建一个CronJob：

```
kubectl run hello --schedule="*/1 * * * *" --restart=OnFailure --image=busybox -- /bin/sh -c "date; echo Hello from the Kubernetes cluster"
```

```
$ kubectl get cronjob
NAME      SCHEDULE      SUSPEND   ACTIVE    LAST-SCHEDULE
hello     */1 * * * *   False     0         <none>
$ kubectl get jobs
NAME                  DESIRED   SUCCESSFUL   AGE
hello-1202039034      1         1            49s
$ pods=$(kubectl get pods --selector=job-name=hello-1202039034 --output=jsonpath={.items..metadata.name} -a)
$ kubectl logs $pods
Mon Aug 29 21:34:09 UTC 2016
Hello from the Kubernetes cluster

# 注意，删除cronjob的时候不会自动删除job，这些job可以用kubectl delete job来删除
$ kubectl delete cronjob hello
cronjob "hello" deleted
```

Cron Job 限制

Cron Job 在每次调度运行时间内 大概 会创建一个 Job 对象。我们之所以说 大概，是因为在特定的环境下可能会创建两个 Job，或者一个 Job 都没创建。我们尝试少发生这种情况，但却不能完全避免。因此，创建 Job 操作应该是 幂等的。

Job 根据它所创建的 Pod 的并行度，负责重试创建 Pod，并就决定这一组 Pod 的成功或失败。Cron Job 根本就不会去检查 Pod。

删除 Cron Job

一旦不再需要 Cron Job，简单地可以使用 `kubectl` 命令删除它：

```
$ kubectl delete cronjob hello
cronjob "hello" deleted
```

这将会终止正在创建的 Job。然而，运行中的 Job 将不会被终止，不会删除 Job 或它们的 Pod。为了清理那些 Job 和 Pod，需要列出该 Cron Job 创建的全部 Job，然后删除它们：

```
$ kubectl get jobs
NAME               DESIRED   SUCCESSFUL   AGE
hello-1201907962   1         1            11m
hello-1202039034   1         1            8m
...

$ kubectl delete jobs hello-1201907962 hello-1202039034 ...
job "hello-1201907962" deleted
job "hello-1202039034" deleted
...
```

一旦 Job 被删除，由 Job 创建的 Pod 也会被删除。注意，所有由名称为 “hello” 的 Cron Job 创建的 Job 会以前缀字符串 “hello-” 进行命名。如果想要删除当前 Namespace 中的所有 Job，可以通过命令 `kubectl delete jobs --all` 立刻删除它们。

Horizontal Pod Autoscaling

应用的资源使用率通常都有高峰和低谷的时候，如何削峰填谷，提高集群的整体资源利用率，让service中的Pod个数自动调整呢？这就有赖于Horizontal Pod Autoscaling了，顾名思义，使Pod水平自动缩放。这个Object（跟Pod、Deployment一样都是API resource）也是最能体现kubernetes之于传统运维价值的地方，不再需要手动扩容了，终于实现自动化了，还可以自定义指标，没准未来还可以通过人工智能自动进化呢！

HPA属于Kubernetes中的**autoscaling** SIG（Special Interest Group），其下有两个feature：

- Arbitrary/Custom Metrics in the Horizontal Pod Autoscaler#117
- Monitoring Pipeline Metrics HPA API #118

Kubernetes自1.2版本引入HPA机制，到1.6版本之前一直是通过kubelet来获取监控指标来判断是否需要扩缩容，1.6版本之后必须通过API server、Heapseter或者kube-aggregator来获取监控指标。

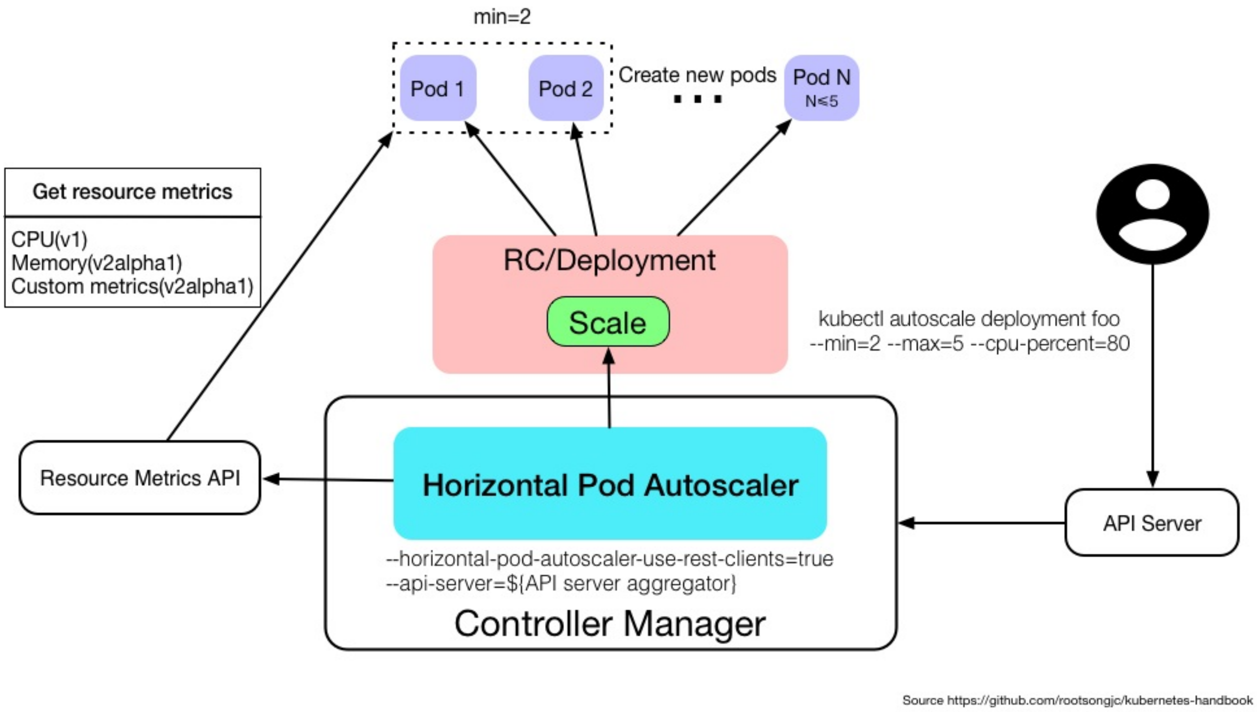
对于1.6以前版本中开启自定义HPA请参考Kubernetes autoscaling based on custom metrics without using a host port，对于1.7及以上版本请参考Configure Kubernetes Autoscaling With Custom Metrics in Kubernetes 1.7 - Bitnami。

HPA解析

Horizontal Pod Autoscaling仅适用于Deployment和ReplicationController，在V1版本中仅支持根据Pod的CPU利用率扩所容，在v1alpha版本中，支持根据内存和用户自定义的metric扩缩容。

如果你不想看下面的文章可以直接看下面的示例图，组件交互、组件的配置、命令示例，都画在图上了。

Horizontal Pod Autoscaling由API server和controller共同实现。



图片 - *horizontal-pod-autoscaler*

Metrics支持

在不同版本的API中，HPA autoscale时可以根据以下指标来判断：

- autoscaling/v1
  - CPU
- autoscaling/v2alpha1
  - 内存
  - 自定义metrics
    - kubernetes1.6起支持自定义metrics，但是必须在kube-controller-manager中配置如下两项：
      - `--horizontal-pod-autoscaler-use-rest-clients=true`
      - `--api-server` 指向kube-aggregator，也可以使用heapster来实现，通过在启动heapster的时候指定 `--api-server=true` 。查看kubernetes metrics
  - 多种metrics组合
    - HPA会根据每个metric的值计算出scale的值，并将最大的那个指作为扩容的最终结果。

使用kubectl管理

Horizontal Pod Autoscaling作为API resource也可以像Pod、Deployment一样使用kubeclt命令管理，使用方法跟它们一样，资源名称为 `hpa` 。

```
kubectl create hpa
kubectl get hpa
kubectl describe hpa
kubectl delete hpa
```

有一点不同的是，可以直接使用 `kubectl autoscale` 直接通过命令行的方式创建Horizontal Pod Autoscaler。

用法如下：

```
kubectl autoscale (-f FILENAME | TYPE NAME | TYPE/NAME) [--min=M
INPODS] --max=MAXPODS
[--cpu-percent=CPU] [flags] [options]
```

举个例子：

```
kubectl autoscale deployment foo --min=2 --max=5 --cpu-percent=8
0
```

为Deployment foo创建 一个autoscaler，当Pod的CPU利用率达到80%的时候，RC的replica数在2到5之间。该命令的详细使用文档见https://kubernetes.io/docs/user-guide/kubectl/v1.6/#autoscale 。

注意 ：如果为ReplicationController创建HPA的话，无法使用rolling update，但是对于Deployment来说是可以的，因为Deployment在执行rolling update的时候会自动创建新的ReplicationController。

什么是 Horizontal Pod Autoscaling？

利用 Horizontal Pod Autoscaling，kubernetes 能够根据监测到的 CPU 利用率（或者在 alpha 版本中支持的应用提供的 metric）自动的扩容 replication controller，deployment 和 replica set。

Horizontal Pod Autoscaler 作为 kubernetes API resource 和 controller 的实现。Resource 确定 controller 的行为。Controller 会根据监测到用户指定的目标的 CPU 利用率周期性得调整 replication controller 或 deployment 的 replica 数量。

Horizontal Pod Autoscaler 如何工作？

Horizontal Pod Autoscaler 由一个控制循环实现，循环周期由 controller manager 中的 `--horizontal-pod-autoscaler-sync-period` 标志指定（默认是 30 秒）。

在每个周期内，controller manager 会查询 HorizontalPodAutoscaler 中定义的 metric 的资源利用率。Controller manager 从 resource metric API（每个 pod 的 resource metric）或者自定义 metric API（所有的metric）中获取 metric。

- 每个 Pod 的 resource metric（例如 CPU），controller 通过 resource metric API 获取 HorizontalPodAutoscaler 中定义的每个 Pod 中的 metric。然后，如果设置了目标利用率，controller 计算利用的值与每个 Pod 的容器里的 resource request 值的百分比。如果设置了目标原始值，将直接使用该原始 metric 值。然后 controller 计算所有目标 Pod 的利用率或原始值（取决于所指定的目标类型）的平均值，产生一个用于缩放所需 replica 数量的比率。 请注意，如果某些 Pod 的容器没有设置相关的 resource request ，则不会定义 Pod 的 CPU 利用率，并且 Aucoscaler 也不会对该 metric 采取任何操作。 有关自动缩放算法如何工作的更多细节，请参阅 自动缩放算法设计文档。
- 对于每个 Pod 自定义的 metric，controller 功能类似于每个 Pod 的 resource metric，只是它使用原始值而不是利用率值。
- 对于 object metric，获取单个度量（描述有问题的对象），并与目标值进行比较，以产生如上所述的比率。

HorizontalPodAutoscaler 控制器可以以两种不同的方式获取 metric ：直接的 Heapster 访问和 REST 客户端访问。

当使用直接的 Heapster 访问时，HorizontalPodAutoscaler 直接通过 API 服务器的服务代理子资源查询 Heapster。需要在集群上部署 Heapster 并在 kube-system namespace 中运行。

有关 REST 客户端访问的详细信息，请参阅 支持自定义度量。

Autoscaler 访问相应的 replication controller，deployment 或 replica set 来缩放子资源。

Scale 是一个允许您动态设置副本数并检查其当前状态的接口。

有关缩放子资源的更多细节可以在 这里 找到。

API Object

Horizontal Pod Autoscaler 是 kubernetes 的 `autoscaling` API 组中的 API 资源。当前的稳定版本中，只支持 CPU 自动扩缩容，可以在 `autoscaling/v1` API 版本中找到。

在 alpha 版本中支持根据内存和自定义 metric 扩缩容，可以在 `autoscaling/v2alpha1` 中找到。`autoscaling/v2alpha1` 中引入的新字段在 `autoscaling/v1` 中是做为 annotation 而保存的。

关于该 API 对象的更多信息，请参阅 HorizontalPodAutoscaler Object。

在 kubectl 中支持 Horizontal Pod Autoscaling

Horizontal Pod Autoscaler 和其他的所有 API 资源一样，通过 `kubectl` 以标准的方式支持。

我们可以使用 `kubectl create` 命令创建一个新的 autoscaler。

我们可以使用 `kubectl get hpa` 列出所有的 autoscaler，使用 `kubectl describe hpa` 获取其详细信息。

最后我们可以使用 `kubectl delete hpa` 删除 autoscaler。

另外，可以使用 `kubectl autoscale` 命令，很轻易的就可以创建一个 Horizontal Pod Autoscaler。

例如，执行 `kubectl autoscale rc foo —min=2 —max=5 —cpu-percent=80` 命令将为 replication controller *foo* 创建一个 autoscaler，目标的 CPU 利用率是 `80%`，replica 的数量介于 2 和 5 之间。

关于 `kubectl autoscale` 的更多信息请参阅 这里。

滚动更新期间的自动扩缩容

目前在Kubernetes中，可以通过直接管理 replication controller 或使用 deployment 对象来执行 滚动更新，该 deployment 对象为您管理基础 replication controller。

Horizontal Pod Autoscaler 仅支持后一种方法：Horizontal Pod Autoscaler 被绑定到 deployment 对象，它设置 deployment 对象的大小，deployment 负责设置底层 replication controller 的大小。

Horizontal Pod Autoscaler 不能使用直接操作 replication controller 进行滚动更新，即不能将 Horizontal Pod Autoscaler 绑定到 replication controller，并进行滚动更新（例如使用 `kubectl rolling-update`）。

这不行的原因是，当滚动更新创建一个新的 replication controller 时，Horizontal Pod Autoscaler 将不会绑定到新的 replication controller 上。

支持多个 metric

Kubernetes 1.6 中增加了支持基于多个 metric 的扩缩容。您可以使用 `autoscaling/v2alpha1` API 版本来为 Horizontal Pod Autoscaler 指定多个 metric。然后 Horizontal Pod Autoscaler controller 将权衡每一个 metric，并根据该 metric 提议一个新的 scale。在所有提议里最大的那个 scale 将作为最终的 scale。

支持自定义 metric

注意： Kubernetes 1.2 根据特定于应用程序的 metric ，通过使用特殊注释的方式，增加了对缩放的 alpha 支持。

在 Kubernetes 1.6中删除了对这些注释的支持，有利于 `autoscaling/v2alpha1` API。 虽然旧的收集自定义 metric 的旧方法仍然可用，但是这些 metric 将不可供 Horizontal Pod Autoscaler 使用，并且用于指定要缩放的自定义 metric 的以前的注释也不在受 Horizontal Pod Autoscaler 认可。

Kubernetes 1.6增加了在 Horizontal Pod Autoscale r中使用自定义 metric 的支持。

您可以为 `autoscaling/v2alpha1` API 中使用的 Horizontal Pod Autoscaler 添加自定义 metric 。

Kubernetes 然后查询新的自定义 metric API 来获取相应自定义 metric 的值。

前提条件

为了在 Horizontal Pod Autoscaler 中使用自定义 metric，您必须在您集群的 controller manager 中将 `--horizontal-pod-autoscaler-use-rest-clients` 标志设置为 true。然后，您必须通过将 controller manager 的目标 API server 设置为 API server aggregator（使用 `--apiserver` 标志），配置您的 controller manager 通过 API server aggregator 与API server 通信。 Resource metric API和自定义 metric API 也必须向 API server aggregator 注册，并且必须由集群上运行的 API server 提供。

您可以使用 Heapster 实现 resource metric API，方法是将 `--api-server` 标志设置为 true 并运行 Heapster。 单独的组件必须提供自定义 metric API（有关自定义metric API的更多信息，可从 k8s.io/metrics repository 获得）。

进一步阅读

- 设计文档：Horizontal Pod Autoscaling
- kubectl autoscale 命令：kubectl autoscale
- 使用例子： Horizontal Pod Autoscaler

参考

HPA设计文档：https://github.com/kubernetes/community/blob/master/contributors/design-proposals/horizontal-pod-autoscaler.md

HPA说明：https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/

HPA详解：https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale-walkthrough/

kubectl autoscale命令详细使用说明：https://kubernetes.io/docs/user-guide/kubectl/v1.6/#autoscale

自定义metrics开发：https://github.com/kubernetes/metrics

1.7版本的kubernetes中启用自定义HPA：Configure Kubernetes Autoscaling With Custom Metrics in Kubernetes 1.7 - Bitnami

自定义指标HPA

Kubernetes中不仅支持CPU、内存为指标的HPA，还支持自定义指标的HPA，例如QPS。

本文中使用的yaml文件见manifests/HPA。

设置自定义指标

kubernetes1.6

> 在kubernetes1.6集群中配置自定义指标的HPA的说明已废弃。

在设置定义指标HPA之前需要先进行如下配置：

- 将heapster的启动参数 `--api-server` 设置为 true

- 启用custom metric API
- 将kube-controller-manager的启动参数中 `--horizontal-pod-autoscaler-use-rest-clients` 设置为true，并指定 `--master` 为API server地址，如 `--master=http://172.20.0.113:8080`

在kubernetes1.5以前很容易设置，参考1.6以前版本的kubernetes中开启自定义HPA，而在1.6中因为取消了原来的annotation方式设置custom metric，只能通过API server和kube-aggregator来获取custom metric，因为只有两种方式来设置了，一是直接通过API server获取heapster的metrics，二是部署kube-aggragator来实现。

我们将在kubernetes1.8版本的kubernetes中，使用聚合的API server来实现自定义指标的HPA。

kuberentes1.7+

确认您的kubernetes版本在1.7或以上，修改以下配置：

- 将kube-controller-manager的启动参数中 `--horizontal-pod-autoscaler-use-rest-clients` 设置为true，并指定 `--master` 为API server地址，如`--master=http://172.20.0.113:8080
- 修改kube-apiserver的配置文件apiserver，增加一条配置 `--requestheader-`

`client-ca-file=/etc/kubernetes/ssl/ca.pem --requestheader-allowed-names=aggregator --requestheader-extra-headers-prefix=X-Remote-Extra- --requestheader-group-headers=X-Remote-Group --requestheader-username-headers=X-Remote-User --proxy-client-cert-file=/etc/kubernetes/ssl/kubernetes.pem --proxy-client-key-file=/etc/kubernetes/ssl/kubernetes-key.pem`，用来配置aggregator的CA证书。

已经内置了 `apiregistration.k8s.io/v1beta1` API，可以直接定义APIService，如：

```yaml
apiVersion: apiregistration.k8s.io/v1
kind: APIService
metadata:
  name: v1.custom-metrics.metrics.k8s.io
spec:
  insecureSkipTLSVerify: true
  group: custom-metrics.metrics.k8s.io
  groupPriorityMinimum: 1000
  versionPriority: 5
  service:
    name: api
    namespace: custom-metrics
  version: v1alpha1
```

部署Prometheus

使用prometheus-operator.yaml文件部署Prometheus operator。

注意：将镜像修改为你自己的镜像仓库地址。

这产生一个自定义的API：http://172.20.0.113:8080/apis/custom-metrics.metrics.k8s.io/v1alpha1，可以通过浏览器访问，还可以使用下面的命令可以检查该API：

```
$ kubectl get --raw=apis/custom-metrics.metrics.k8s.io/v1alpha1
{"kind":"APIResourceList","apiVersion":"v1","groupVersion":"cust
om-metrics.metrics.k8s.io/v1alpha1","resources":[{"name":"jobs.b
atch/http_requests","singularName":"","namespaced":true,"kind":"
MetricValueList","verbs":["get"]},{"name":"namespaces/http_reque
sts","singularName":"","namespaced":false,"kind":"MetricValueLis
t","verbs":["get"]},{"name":"jobs.batch/up","singularName":"","n
amespaced":true,"kind":"MetricValueList","verbs":["get"]},{"name"
:"pods/up","singularName":"","namespaced":true,"kind":"MetricVal
ueList","verbs":["get"]},{"name":"services/scrape_samples_scrape
d","singularName":"","namespaced":true,"kind":"MetricValueList",
"verbs":["get"]},{"name":"namespaces/scrape_samples_scraped","si
ngularName":"","namespaced":false,"kind":"MetricValueList","verb
s":["get"]},{"name":"pods/scrape_duration_seconds","singularName"
:"","namespaced":true,"kind":"MetricValueList","verbs":["get"]},{
"name":"services/scrape_duration_seconds","singularName":"","nam
espaced":true,"kind":"MetricValueList","verbs":["get"]},{"name":
"pods/http_requests","singularName":"","namespaced":true,"kind":
"MetricValueList","verbs":["get"]},{"name":"pods/scrape_samples_
post_metric_relabeling","singularName":"","namespaced":true,"kin
d":"MetricValueList","verbs":["get"]},{"name":"jobs.batch/scrape
_samples_scraped","singularName":"","namespaced":true,"kind":"Me
tricValueList","verbs":["get"]},{"name":"jobs.batch/scrape_durat
ion_seconds","singularName":"","namespaced":true,"kind":"MetricV
alueList","verbs":["get"]},{"name":"namespaces/scrape_duration_s
econds","singularName":"","namespaced":false,"kind":"MetricValue
List","verbs":["get"]},{"name":"namespaces/scrape_samples_post_m
etric_relabeling","singularName":"","namespaced":false,"kind":"M
etricValueList","verbs":["get"]},{"name":"services/scrape_sample
s_post_metric_relabeling","singularName":"","namespaced":true,"k
ind":"MetricValueList","verbs":["get"]},{"name":"services/up","s
ingularName":"","namespaced":true,"kind":"MetricValueList","verb
s":["get"]},{"name":"pods/scrape_samples_scraped","singularName":
"","namespaced":true,"kind":"MetricValueList","verbs":["get"]},{
"name":"services/http_requests","singularName":"","namespaced":t
rue,"kind":"MetricValueList","verbs":["get"]},{"name":"jobs.batc
h/scrape_samples_post_metric_relabeling","singularName":"","name
spaced":true,"kind":"MetricValueList","verbs":["get"]},{"name":"
namespaces/up","singularName":"","namespaced":false,"kind":"Metr
icValueList","verbs":["get"]}]}
```

参考

1.6以前版本的kubernetes中开启自定义HPA

1.7版本的kubernetes中启用自定义HPA

Horizontal Pod Autoscaler Walkthrough

Kubernetes 1.8: Now with 100% Daily Value of Custom Metrics

Arbitrary/Custom Metrics in the Horizontal Pod Autoscaler#117

服务发现

Kubernetes中为了实现服务实例间的负载均衡和不同服务间的服务发现，创造了Serivce对象，同时又为从集群外部访问集群创建了Ingress对象。

Service

Kubernetes `Pod` 是有生命周期的，它们可以被创建，也可以被销毁，然而一旦被销毁生命就永远结束。 通过 `ReplicationController` 能够动态地创建和销毁 `Pod` （例如，需要进行扩缩容，或者执行 滚动升级）。 每个 `Pod` 都会获取它自己的 IP 地址，即使这些 IP 地址不总是稳定可依赖的。 这会导致一个问题：在 Kubernetes 集群中，如果一组 `Pod` （称为 backend）为其它 `Pod` （称为 frontend）提供服务，那么那些 frontend 该如何发现，并连接到这组 `Pod` 中的哪些 backend 呢？

关于 `Service`

Kubernetes `Service` 定义了这样一种抽象：一个 `Pod` 的逻辑分组，一种可以访问它们的策略 —— 通常称为微服务。 这一组 `Pod` 能够被 `Service` 访问到，通常是通过 `Label Selector` （查看下面了解，为什么可能需要没有 selector 的 `Service` ）实现的。

举个例子，考虑一个图片处理 backend，它运行了3个副本。这些副本是可互换的 —— frontend 不需要关心它们调用了哪个 backend 副本。 然而组成这一组 backend 程序的 `Pod` 实际上可能会发生变化，frontend 客户端不应该也没必要知道，而且也不需要跟踪这一组 backend 的状态。 `Service` 定义的抽象能够解耦这种关联。

对 Kubernetes 集群中的应用，Kubernetes 提供了简单的 `Endpoints` API，只要 `Service` 中的一组 `Pod` 发生变更，应用程序就会被更新。 对非 Kubernetes 集群中的应用，Kubernetes 提供了基于 VIP 的网桥的方式访问 `Service` ，再由 `Service` 重定向到 backend `Pod` 。

定义 Service

一个 `Service` 在 Kubernetes 中是一个 REST 对象，和 `Pod` 类似。 像所有的 REST 对象一样， `Service` 定义可以基于 POST 方式，请求 apiserver 创建新的实例。 例如，假定有一组 `Pod` ，它们对外暴露了 9376 端口，同时还被打上 `"app=MyApp"` 标签。

```yaml
kind: Service
apiVersion: v1
metadata:
  name: my-service
spec:
  selector:
    app: MyApp
  ports:
    - protocol: TCP
      port: 80
      targetPort: 9376
```

上述配置将创建一个名称为 “my-service” 的 `Service` 对象，它会将请求代理到使用 TCP 端口 9376，并且具有标签 `"app=MyApp"` 的 `Pod` 上。 这个 `Service` 将被指派一个 IP 地址（通常称为 “Cluster IP”），它会被服务的代理使用（见下面）。 该 `Service` 的 selector 将会持续评估，处理结果将被 POST 到一个名称为 “my-service” 的 `Endpoints` 对象上。

需要注意的是， `Service` 能够将一个接收端口映射到任意的 `targetPort` 。 默认情况下， `targetPort` 将被设置为与 `port` 字段相同的值。 可能更有趣的是， `targetPort` 可以是一个字符串，引用了 backend `Pod` 的一个端口的名称。 但是，实际指派给该端口名称的端口号，在每个 backend `Pod` 中可能并不相同。 对于部署和设计 `Service` ，这种方式会提供更大的灵活性。 例如，可以在 backend 软件下一个版本中，修改 Pod 暴露的端口，并不会中断客户端的调用。

Kubernetes `Service` 能够支持 `TCP` 和 `UDP` 协议，默认 `TCP` 协议。

没有 selector 的 Service

Service 抽象了该如何访问 Kubernetes `Pod` ，但也能够抽象其它类型的 backend，例如：

- 希望在生产环境中使用外部的数据库集群，但测试环境使用自己的数据库。
- 希望服务指向另一个 `Namespace` 中或其它集群中的服务。
- 正在将工作负载转移到 Kubernetes 集群，和运行在 Kubernetes 集群之外的 backend。

在任何这些场景中，都能够定义没有 selector 的 `Service` ：

```yaml
kind: Service
apiVersion: v1
metadata:
  name: my-service
spec:
  ports:
    - protocol: TCP
      port: 80
      targetPort: 9376
```

由于这个 `Service` 没有 selector，就不会创建相关的 `Endpoints` 对象。可以手动将 `Service` 映射到指定的 `Endpoints` ：

```yaml
kind: Endpoints
apiVersion: v1
metadata:
  name: my-service
subsets:
  - addresses:
      - ip: 1.2.3.4
    ports:
      - port: 9376
```

注意：Endpoint IP 地址不能是 loopback（127.0.0.0/8）、 link-local（169.254.0.0/16）、或者 link-local 多播（224.0.0.0/24）。

访问没有 selector 的 `Service` ，与有 selector 的 `Service` 的原理相同。请求将被路由到用户定义的 Endpoint（该示例中为 `1.2.3.4:9376` ）。

ExternalName `Service` 是 `Service` 的特例，它没有 selector，也没有定义任何的端口和 Endpoint。 相反地，对于运行在集群外部的服务，它通过返回该外部服务的别名这种方式来提供服务。

```yaml
kind: Service
apiVersion: v1
metadata:
  name: my-service
  namespace: prod
spec:
  type: ExternalName
  externalName: my.database.example.com
```

当查询主机 `my-service.prod.svc.CLUSTER` 时，集群的 DNS 服务将返回一个值为 `my.database.example.com` 的 `CNAME` 记录。 访问这个服务的工作方式与其它的相同，唯一不同的是重定向发生在 DNS 层，而且不会进行代理或转发。如果后续决定要将数据库迁移到 Kubernetes 集群中，可以启动对应的 Pod，增加合适的 Selector 或 Endpoint，修改 `Service` 的 `type` 。

VIP 和 Service 代理

在 Kubernetes 集群中，每个 Node 运行一个 `kube-proxy` 进程。 `kube-proxy` 负责为 `Service` 实现了一种 VIP（虚拟 IP）的形式，而不是 `ExternalName` 的形式。 在 Kubernetes v1.0 版本，代理完全在 userspace。在 Kubernetes v1.1 版本，新增了 iptables 代理，但并不是默认的运行模式。 从 Kubernetes v1.2 起，默认就是 iptables 代理。

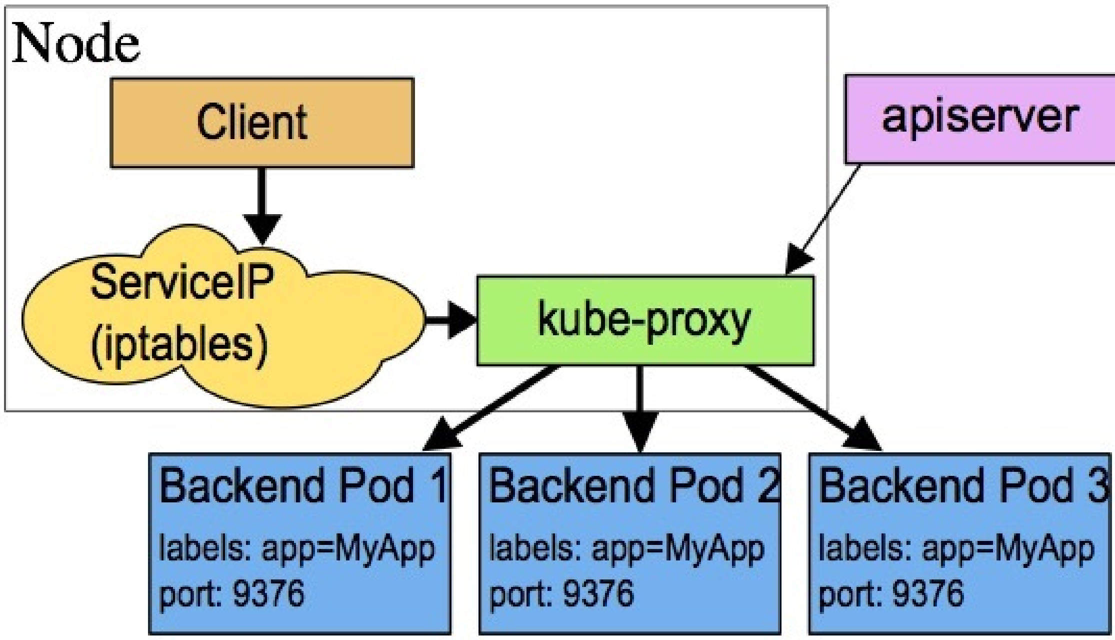
在 Kubernetes v1.0 版本， `Service` 是 “4层”（TCP/UDP over IP）概念。 在 Kubernetes v1.1 版本，新增了 `Ingress` API（beta 版），用来表示 “7层”（HTTP）服务。

userspace 代理模式

这种模式，kube-proxy 会监视 Kubernetes master 对 `Service` 对象和 `Endpoints` 对象的添加和移除。 对每个 `Service` ，它会在本地 Node 上打开一个端口（随机选择）。 任何连接到“代理端口”的请求，都会被代理到 `Service` 的backend `Pods` 中的某个上面（如 `Endpoints` 所报告的一样）。 使用哪个 backend `Pod` ，是基于 `Service` 的 `SessionAffinity` 来确定的。 最后，它安装 iptables 规则，捕获到达该 `Service` 的 `clusterIP` （是虚拟 IP）和 `Port` 的请求，并重定向到代理端口，代理端口再代理请求到 backend `Pod` 。

网络返回的结果是，任何到达 `Service` 的 IP:Port 的请求，都会被代理到一个合适的 backend，不需要客户端知道关于 Kubernetes、 `Service` 、或 `Pod` 的任何信息。

默认的策略是，通过 round-robin 算法来选择 backend `Pod` 。 实现基于客户端 IP 的会话亲和性，可以通过设置 `service.spec.sessionAffinity` 的值为 `"ClientIP"` （默认值为 `"None"` ）。



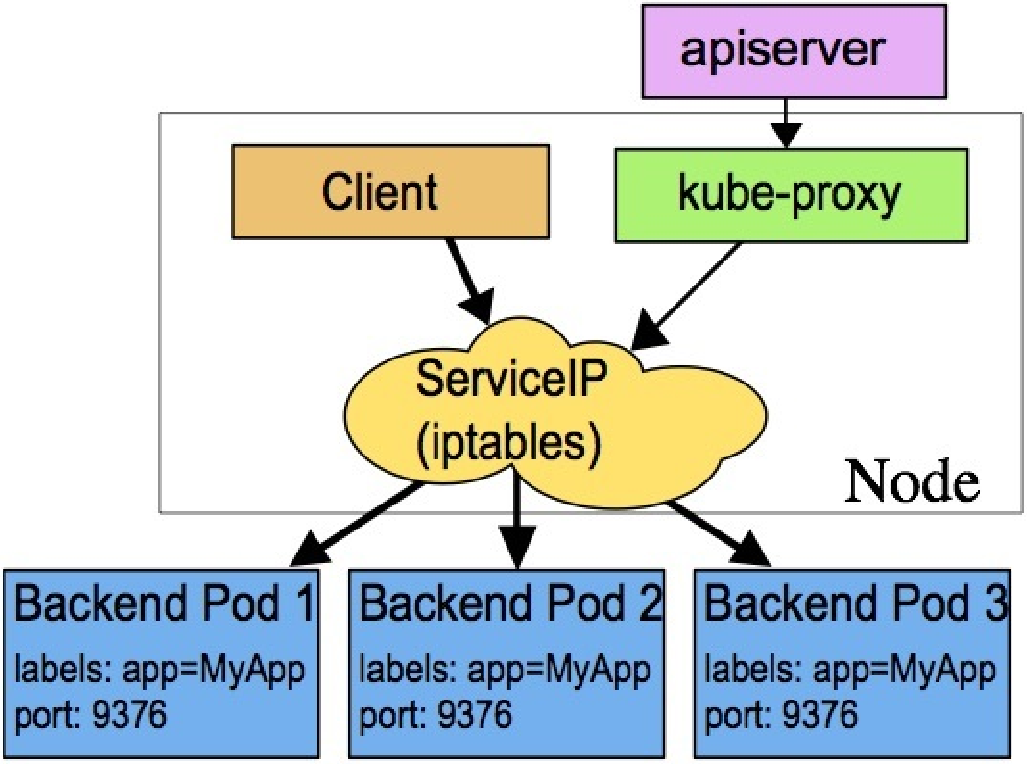
图片 - *userspace*代理模式下*Service*概览图

iptables 代理模式

这种模式，kube-proxy 会监视 Kubernetes master 对 `Service` 对象和 `Endpoints` 对象的添加和移除。 对每个 `Service`，它会安装 iptables 规则，从而捕获到达该 `Service` 的 `clusterIP`（虚拟 IP）和端口的请求，进而将请求重定向到 `Service` 的一组 backend 中的某个上面。 对于每个 `Endpoints` 对象，它也会安装 iptables 规则，这个规则会选择一个 backend `Pod`。

默认的策略是，随机选择一个 backend。 实现基于客户端 IP 的会话亲和性，可以将 `service.spec.sessionAffinity` 的值设置为 `"ClientIP"` （默认值为 `"None"`）。

和 userspace 代理类似，网络返回的结果是，任何到达 `Service` 的 IP:Port 的请求，都会被代理到一个合适的 backend，不需要客户端知道关于 Kubernetes、`Service`、或 `Pod` 的任何信息。 这应该比 userspace 代理更快、更可靠。然而，不像 userspace 代理，如果初始选择的 `Pod` 没有响应，iptables 代理不能自动地重试另一个 `Pod`，所以它需要依赖 readiness probes。



图片 - *iptables*代理模式下*Service*概览图

多端口 Service

很多 `Service` 需要暴露多个端口。对于这种情况，Kubernetes 支持在 `Service` 对象中定义多个端口。 当使用多个端口时，必须给出所有的端口的名称，这样 Endpoint 就不会产生歧义，例如：

```
kind: Service
apiVersion: v1
metadata:
  name: my-service
spec:
    selector:
      app: MyApp
    ports:
      - name: http
        protocol: TCP
        port: 80
        targetPort: 9376
      - name: https
        protocol: TCP
        port: 443
        targetPort: 9377
```

选择自己的 IP 地址

在 `Service` 创建的请求中，可以通过设置 `spec.clusterIP` 字段来指定自己的集群 IP 地址。 比如，希望替换一个已经已存在的 DNS 条目，或者遗留系统已经配置了一个固定的 IP 且很难重新配置。 用户选择的 IP 地址必须合法，并且这个 IP 地址在 `service-cluster-ip-range` CIDR 范围内，这对 API Server 来说是通过一个标识来指定的。 如果 IP 地址不合法，API Server 会返回 HTTP 状态码 422，表示值不合法。

为何不使用 round-robin DNS？

一个不时出现的问题是，为什么我们都使用 VIP 的方式，而不使用标准的 round-robin DNS，有如下几个原因：

- 长久以来，DNS 库都没能认真对待 DNS TTL、缓存域名查询结果
- 很多应用只查询一次 DNS 并缓存了结果
  - 就算应用和库能够正确查询解析，每个客户端反复重解析造成的负载也是非常难以管理的

我们尽力阻止用户做那些对他们没有好处的事情，如果很多人都来问这个问题，我们可能会选择实现它。

服务发现

Kubernetes 支持2种基本的服务发现模式 —— 环境变量和 DNS。

环境变量

当 `Pod` 运行在 `Node` 上，kubelet 会为每个活跃的 `Service` 添加一组环境变量。 它同时支持 Docker links 兼容 变量（查看 makeLinkVariables）、简单的 `{SVCNAME}_SERVICE_HOST` 和 `{SVCNAME}_SERVICE_PORT` 变量，这里 `Service` 的名称需大写，横线被转换成下划线。

举个例子，一个名称为 `"redis-master"` 的 Service 暴露了 TCP 端口 6379，同时给它分配了 Cluster IP 地址 10.0.0.11，这个 Service 生成了如下环境变量：

```
REDIS_MASTER_SERVICE_HOST=10.0.0.11
REDIS_MASTER_SERVICE_PORT=6379
REDIS_MASTER_PORT=tcp://10.0.0.11:6379
REDIS_MASTER_PORT_6379_TCP=tcp://10.0.0.11:6379
REDIS_MASTER_PORT_6379_TCP_PROTO=tcp
REDIS_MASTER_PORT_6379_TCP_PORT=6379
REDIS_MASTER_PORT_6379_TCP_ADDR=10.0.0.11
```

这意味着需要有顺序的要求—— `Pod` 想要访问的任何 `Service` 必须在 `Pod` 自己之前被创建，否则这些环境变量就不会被赋值。DNS 并没有这个限制。

DNS

一个可选（尽管强烈推荐）集群插件 是 DNS 服务器。 DNS 服务器监视着创建新 `Service` 的 Kubernetes API，从而为每一个 `Service` 创建一组 DNS 记录。如果整个集群的 DNS 一直被启用，那么所有的 `Pod` 应该能够自动对 `Service` 进行名称解析。

例如，有一个名称为 `"my-service"` 的 `Service`，它在 Kubernetes 集群中名为 `"my-ns"` 的 `Namespace` 中，为 `"my-service.my-ns"` 创建了一条 DNS 记录。 在名称为 `"my-ns"` 的 `Namespace` 中的 `Pod` 应该能够简单地通过名称查询找到 `"my-service"`。 在另一个 `Namespace` 中的 `Pod` 必须限定名称为 `"my-service.my-ns"`。 这些名称查询的结果是 Cluster IP。

Kubernetes 也支持对端口名称的 DNS SRV（Service）记录。 如果名称为 `"my-service.my-ns"` 的 `Service` 有一个名为 `"http"` 的 `TCP` 端口，可以对 `"_http._tcp.my-service.my-ns"` 执行 DNS SRV 查询，得到 `"http"` 的端口号。

Kubernetes DNS 服务器是唯一的一种能够访问 `ExternalName` 类型的 Service 的方式。 更多信息可以查看 DNS Pod 和 Service。

Headless Service

有时不需要或不想要负载均衡，以及单独的 Service IP。 遇到这种情况，可以通过指定 Cluster IP（ `spec.clusterIP` ）的值为 `"None"` 来创建 `Headless` Service。

这个选项允许开发人员自由寻找他们自己的方式，从而降低与 Kubernetes 系统的耦合性。 应用仍然可以使用一种自注册的模式和适配器，对其它需要发现机制的系统能够很容易地基于这个 API 来构建。

对这类 `Service` 并不会分配 Cluster IP，kube-proxy 不会处理它们，而且平台也不会为它们进行负载均衡和路由。 DNS 如何实现自动配置，依赖于 `Service` 是否定义了 selector。

配置 Selector

对定义了 selector 的 Headless Service，Endpoint 控制器在 API 中创建了 `Endpoints` 记录，并且修改 DNS 配置返回 A 记录（地址），通过这个地址直接到达 `Service` 的后端 `Pod` 上。

不配置 Selector

对没有定义 selector 的 Headless Service，Endpoint 控制器不会创建 `Endpoints` 记录。 然而 DNS 系统会查找和配置，无论是：

- `ExternalName` 类型 Service 的 CNAME 记录
  - 记录：与 Service 共享一个名称的任何 `Endpoints` ，以及所有其它类型

发布服务 —— 服务类型

对一些应用（如 Frontend）的某些部分，可能希望通过外部（Kubernetes 集群外部）IP 地址暴露 Service。

Kubernetes `ServiceTypes` 允许指定一个需要的类型的 Service，默认是 `ClusterIP` 类型。

`Type` 的取值以及行为如下：

- `ClusterIP` ：通过集群的内部 IP 暴露服务，选择该值，服务只能够在集群内部可以访问，这也是默认的 `ServiceType` 。
- `NodePort` ：通过每个 Node 上的 IP 和静态端口（ `NodePort` ）暴露服务。 `NodePort` 服务会路由到 `ClusterIP` 服务，这个 `ClusterIP` 服务会自动创建。通过请求 `<NodeIP>:<NodePort>` ，可以从集群的外部访问一个 `NodePort` 服务。

- `LoadBalancer`：使用云提供商的负载均衡器，可以向外部暴露服务。外部的负载均衡器可以路由到 `NodePort` 服务和 `ClusterIP` 服务。
- `ExternalName`：通过返回 `CNAME` 和它的值，可以将服务映射到 `externalName` 字段的内容（例如， `foo.bar.example.com`）。 没有任何类型代理被创建，这只有 Kubernetes 1.7 或更高版本的 `kube-dns` 才支持。

NodePort 类型

如果设置 `type` 的值为 `"NodePort"`，Kubernetes master 将从给定的配置范围内（默认：30000-32767）分配端口，每个 Node 将从该端口（每个 Node 上的同一端口）代理到 `Service`。该端口将通过 `Service` 的 `spec.ports[*].nodePort` 字段被指定。

如果需要指定的端口号，可以配置 `nodePort` 的值，系统将分配这个端口，否则调用 API 将会失败（比如，需要关心端口冲突的可能性）。

这可以让开发人员自由地安装他们自己的负载均衡器，并配置 Kubernetes 不能完全支持的环境参数，或者直接暴露一个或多个 Node 的 IP 地址。

需要注意的是，Service 将能够通过 `<NodeIP>:spec.ports[*].nodePort` 和 `spec.clusterIp:spec.ports[*].port` 而对外可见。

LoadBalancer 类型

使用支持外部负载均衡器的云提供商的服务，设置 `type` 的值为 `"LoadBalancer"`，将为 `Service` 提供负载均衡器。 负载均衡器是异步创建的，关于被提供的负载均衡器的信息将会通过 `Service` 的 `status.loadBalancer` 字段被发布出去。

```yaml
kind: Service
apiVersion: v1
metadata:
  name: my-service
spec:
  selector:
    app: MyApp
  ports:
    - protocol: TCP
      port: 80
      targetPort: 9376
      nodePort: 30061
  clusterIP: 10.0.171.239
  loadBalancerIP: 78.11.24.19
  type: LoadBalancer
status:
  loadBalancer:
    ingress:
      - ip: 146.148.47.155
```

来自外部负载均衡器的流量将直接打到 backend `Pod` 上，不过实际它们是如何工作的，这要依赖于云提供商。 在这些情况下，将根据用户设置的 `loadBalancerIP` 来创建负载均衡器。 某些云提供商允许设置 `loadBalancerIP` 。如果没有设置 `loadBalancerIP` ，将会给负载均衡器指派一个临时 IP。 如果设置了 `loadBalancerIP` ，但云提供商并不支持这种特性，那么设置的 `loadBalancerIP` 值将会被忽略掉。

AWS 内部负载均衡器

在混合云环境中，有时从虚拟私有云（VPC）环境中的服务路由流量是非常有必要的。 可以通过在 `Service` 中增加 `annotation` 来实现，如下所示：

```yaml
[...]
metadata:
    name: my-service
    annotations:
        service.beta.kubernetes.io/aws-load-balancer-internal: 0
.0.0.0/0
[...]
```

在水平分割的 DNS 环境中，需要两个 `Service` 来将外部和内部的流量路由到 Endpoint 上。

AWS SSL 支持

对运行在 AWS 上部分支持 SSL 的集群，从 1.3 版本开始，可以为 `LoadBalancer` 类型的 `Service` 增加两个 annotation：

```
    metadata:
      name: my-service
      annotations:
        service.beta.kubernetes.io/aws-load-balancer-ssl-cert: a
rn:aws:acm:us-east-1:123456789012:certificate/12345678-1234-1234
-1234-123456789012
```

第一个 annotation 指定了使用的证书。它可以是第三方发行商发行的证书，这个证书或者被上传到 IAM，或者由 AWS 的证书管理器创建。

```
    metadata:
      name: my-service
      annotations:
         service.beta.kubernetes.io/aws-load-balancer-backend-pr
otocol: (https|http|ssl|tcp)
```

第二个 annotation 指定了 `Pod` 使用的协议。 对于 HTTPS 和 SSL，ELB 将期望该 `Pod` 基于加密的连接来认证自身。

HTTP 和 HTTPS 将选择7层代理：ELB 将中断与用户的连接，当转发请求时，会解析 Header 信息并添加上用户的 IP 地址（ `Pod` 将只能在连接的另一端看到该 IP 地址）。

TCP 和 SSL 将选择4层代理：ELB 将转发流量，并不修改 Header 信息。

外部 IP

如果外部的 IP 路由到集群中一个或多个 Node 上，Kubernetes `Service` 会被暴露给这些 `externalIPs` 。 通过外部 IP（作为目的 IP 地址）进入到集群，打到 `Service` 的端口上的流量，将会被路由到 `Service` 的 Endpoint 上。 `externalIPs` 不会被 Kubernetes 管理，它属于集群管理员的职责范畴。

根据 `Service` 的规定， `externalIPs` 可以同任意的 `ServiceType` 来一起指定。 在下面的例子中， `my-service` 可以在 80.11.12.10:80（外部 IP:端口）上被客户端访问。

```yaml
kind: Service
apiVersion: v1
metadata:
  name: my-service
spec:
  selector:
    app: MyApp
  ports:
    - name: http
      protocol: TCP
      port: 80
      targetPort: 9376
  externalIPs:
    - 80.11.12.10
```

不足之处

为 VIP 使用 userspace 代理，将只适合小型到中型规模的集群，不能够扩展到上千 `Service` 的大型集群。查看 最初设计方案 获取更多细节。

使用 userspace 代理，隐藏了访问 `Service` 的数据包的源 IP 地址。这使得一些类型的防火墙无法起作用。iptables 代理不会隐藏 Kubernetes 集群内部的 IP 地址，但却要求客户端请求必须通过一个负载均衡器或 Node 端口。

`Type` 字段支持嵌套功能 —— 每一层需要添加到上一层里面。不会严格要求所有云提供商（例如，GCE 就没必要为了使一个 `LoadBalancer` 能工作而分配一个 `NodePort`，但是 AWS 需要），但当前 API 是强制要求的。

未来工作

未来我们能预见到，代理策略可能会变得比简单的 round-robin 均衡策略有更多细微的差别，比如 master 选举或分片。我们也能想到，某些 `Service` 将具有 "真正" 的负载均衡器，这种情况下 VIP 将简化数据包的传输。

我们打算为 L7（HTTP） `Service` 改进我们对它的支持。

我们打算为 `Service` 实现更加灵活的请求进入模式，这些 `Service` 包含当前 `ClusterIP` 、 `NodePort` 和 `LoadBalancer` 模式，或者更多。

VIP 的那些骇人听闻的细节

对很多想使用 `Service` 的人来说，前面的信息应该足够了。 然而，有很多内部原理性的内容，还是值去理解的。

避免冲突

Kubernetes 最主要的哲学之一，是用户不应该暴露那些能够导致他们操作失败、但又不是他们的过错的场景。 这种场景下，让我们来看一下网络端口 —— 用户不应该必须选择一个端口号，而且该端口还有可能与其他用户的冲突。 这就是说，在彼此隔离状态下仍然会出现失败。

为了使用户能够为他们的 `Service` 选择一个端口号，我们必须确保不能有2个 `Service` 发生冲突。 我们可以通过为每个 `Service` 分配它们自己的 IP 地址来实现。

为了保证每个 `Service` 被分配到一个唯一的 IP，需要一个内部的分配器能够原子地更新 etcd 中的一个全局分配映射表，这个更新操作要先于创建每一个 `Service` 。 为了使 `Service` 能够获取到 IP，这个映射表对象必须在注册中心存在，否则创建 `Service` 将会失败，指示一个 IP 不能被分配。 一个后台 Controller 的职责是创建映射表（从 Kubernetes 的旧版本迁移过来，旧版本中是通过在内存中加锁的方式实现），并检查由于管理员干预和清除任意 IP 造成的不合理分配，这些 IP 被分配了但当前没有 `Service` 使用它们。

IP 和 VIP

不像 `Pod` 的 IP 地址，它实际路由到一个固定的目的地，`Service` 的 IP 实际上不能通过单个主机来进行应答。 相反，我们使用 `iptables` （Linux 中的数据包处理逻辑）来定义一个虚拟IP地址（VIP），它可以根据需要透明地进行重定向。 当客户端连接到 VIP 时，它们的流量会自动地传输到一个合适的 Endpoint。 环境变量和 DNS，实际上会根据 `Service` 的 VIP 和端口来进行填充。

Userspace

作为一个例子，考虑前面提到的图片处理应用程序。 当创建 backend `Service` 时，Kubernetes master 会给它指派一个虚拟 IP 地址，比如 10.0.0.1。 假设 `Service` 的端口是 1234，该 `Service` 会被集群中所有的 `kube-proxy` 实例观察到。 当代理看到一个新的 `Service` ， 它会打开一个新的端口，建立一个从该 VIP 重定向到新端口的 iptables，并开始接收请求连接。

当一个客户端连接到一个 VIP，iptables 规则开始起作用，它会重定向该数据包到 `Service代理` 的端口。 `Service代理` 选择一个 backend，并将客户端的流量代理到 backend 上。

这意味着 `Service` 的所有者能够选择任何他们想使用的端口，而不存在冲突的风险。 客户端可以简单地连接到一个 IP 和端口，而不需要知道实际访问了哪些 `Pod` 。

Iptables

再次考虑前面提到的图片处理应用程序。 当创建 backend `Service` 时，Kubernetes master 会给它指派一个虚拟 IP 地址，比如 10.0.0.1。 假设 `Service` 的端口是 1234，该 `Service` 会被集群中所有的 `kube-proxy` 实例观察到。 当代理看到一个新的 `Service` ， 它会安装一系列的 iptables 规则，从 VIP 重定向到 per- `Service` 规则。 该 per- `Service` 规则连接到 per- `Endpoint` 规则，该 per- `Endpoint` 规则会重定向（目标 NAT）到 backend。

当一个客户端连接到一个 VIP，iptables 规则开始起作用。一个 backend 会被选择（或者根据会话亲和性，或者随机），数据包被重定向到这个 backend。 不像 userspace 代理，数据包从来不拷贝到用户空间，kube-proxy 不是必须为该 VIP 工作而运行，并且客户端 IP 是不可更改的。 当流量打到 Node 的端口上，或通过负载均衡器，会执行相同的基本流程，但是在那些案例中客户端 IP 是可以更改的。

API 对象

在 Kubernetes REST API 中，Service 是 top-level 资源。关于 API 对象的更多细节可以查看：Service API 对象。

更多信息

- 使用 Service 连接 Frontend 到 Backend

Ingress解析

前言

这是kubernete官方文档中Ingress Resource的翻译，后面的章节会讲到使用Traefik来做Ingress controller，文章末尾给出了几个相关链接。

术语

在本篇文章中你将会看到一些在其他地方被交叉使用的术语，为了防止产生歧义，我们首先来澄清下。

- 节点：Kubernetes集群中的一台物理机或者虚拟机。
- 集群：位于Internet防火墙后的节点，这是kubernetes管理的主要计算资源。
- 边界路由器：为集群强制执行防火墙策略的路由器。 这可能是由云提供商或物理硬件管理的网关。
- 集群网络：一组逻辑或物理链接，可根据Kubernetes网络模型实现群集内的通信。 集群网络的实现包括Overlay模型的 flannel 和基于SDN的OVS。
- 服务：使用标签选择器标识一组pod成为的Kubernetes服务。 除非另有说明，否则服务假定在集群网络内仅可通过虚拟IP访问。

什么是Ingress？

通常情况下，service和pod仅可在集群内部网络中通过IP地址访问。所有到达边界路由器的流量或被丢弃或被转发到其他地方。从概念上讲，可能像下面这样：

```
    internet
        |
  ------------
  [ Services ]
```

Ingress是授权入站连接到达集群服务的规则集合。

```
    internet
        |
   [ Ingress ]
   --|-----|--
   [ Services ]
```

你可以给Ingress配置提供外部可访问的URL、负载均衡、SSL、基于名称的虚拟主机等。用户通过POST Ingress资源到API server的方式来请求ingress。 Ingress controller负责实现Ingress，通常使用负载平衡器，它还可以配置边界路由和其他前端，这有助于以HA方式处理流量。

先决条件

在使用Ingress resource之前，有必要先了解下面几件事情。Ingress是beta版本的resource，在kubernetes1.1之前还没有。你需要一个 `Ingress Controller` 来实现 `Ingress` ，单纯的创建一个 `Ingress` 没有任何意义。

GCE/GKE会在master节点上部署一个ingress controller。你可以在一个pod中部署任意个自定义的ingress controller。你必须正确地annotate每个ingress，比如 运行多个ingress controller 和 关闭glbc.

确定你已经阅读了Ingress controller的beta版本限制。在非GCE/GKE的环境中，你需要在pod中部署一个controller。

Ingress Resource

最简化的Ingress配置：

```
1: apiVersion: extensions/v1beta1
2: kind: Ingress
3: metadata:
4:   name: test-ingress
5: spec:
6:   rules:
7:   - http:
8:       paths:
9:       - path: /testpath
10:         backend:
11:            serviceName: test
12:            servicePort: 80
```

如果你没有配置*Ingress controller*就将其*POST*到*API server*不会有任何用处

配置说明

1-4行：跟Kubernetes的其他配置一样，ingress的配置也需要`apiVersion`，`kind`和`metadata`字段。配置文件的详细说明请查看部署应用, 配置容器和 使用resources.

5-7行: Ingress spec 中包含配置一个loadbalancer或proxy server的所有信息。最重要的是，它包含了一个匹配所有入站请求的规则列表。目前ingress只支持http规则。

8-9行：每条http规则包含以下信息：一个`host`配置项（比如for.bar.com，在这个例子中默认是*），`path`列表（比如：/testpath），每个path都关联一个`backend`(比如test:80)。在loadbalancer将流量转发到backend之前，所有的入站请求都要先匹配host和path。

10-12行：正如 services doc中描述的那样，backend是一个`service:port`的组合。Ingress的流量被转发到它所匹配的backend。

全局参数：为了简单起见，Ingress示例中没有全局参数，请参阅资源完整定义的api参考。 在所有请求都不能跟spec中的path匹配的情况下，请求被发送到Ingress controller的默认后端，可以指定全局缺省backend。

Ingress controllers

为了使Ingress正常工作，集群中必须运行Ingress controller。 这与其他类型的控制器不同，其他类型的控制器通常作为`kube-controller-manager`二进制文件的一部分运行，在集群启动时自动启动。 你需要选择最适合自己集群的Ingress controller或者自己实现一个。 示例和说明可以在这里找到。

在你开始前

以下文档描述了Ingress资源中公开的一组跨平台功能。 理想情况下，所有的Ingress controller都应该符合这个规范，但是我们还没有实现。 GCE和nginx控制器的文档分别在这里和这里。确保您查看控制器特定的文档，以便您了解每个文档的注意事项。

Ingress类型

单Service Ingress

Kubernetes中已经存在一些概念可以暴露单个service（查看替代方案），但是你仍然可以通过Ingress来实现，通过指定一个没有rule的默认backend的方式。

ingress.yaml定义文件：

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: test-ingress
spec:
  backend:
    serviceName: testsvc
    servicePort: 80
```

使用 `kubectl create -f` 命令创建，然后查看ingress：

```
$ kubectl get ing
NAME                RULE          BACKEND        ADDRESS
test-ingress        -             testsvc:80     107.178.254.228
```

`107.178.254.228` 就是Ingress controller为了实现Ingress而分配的IP地址。 `RULE` 列表示所有发送给该IP的流量都被转发到了 `BACKEND` 所列的Kubernetes service上。

简单展开

如前面描述的那样，kubernete pod中的IP只在集群网络内部可见，我们需要在边界设置一个东西，让它能够接收ingress的流量并将它们转发到正确的端点上。这个东西一般是高可用的loadbalancer。使用Ingress能够允许你将loadbalancer的个数降低到最少，例如，假如你想要创建这样的一个设置：

```
foo.bar.com -> 178.91.123.132 -> / foo    s1:80
                                 / bar    s2:80
```

你需要一个这样的ingress：

```yaml
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: test
spec:
  rules:
  - host: foo.bar.com
    http:
      paths:
      - path: /foo
        backend:
          serviceName: s1
          servicePort: 80
      - path: /bar
        backend:
          serviceName: s2
          servicePort: 80
```

使用 `kubectl create -f` 创建完ingress后：

```
$ kubectl get ing
NAME      RULE          BACKEND   ADDRESS
test      -
          foo.bar.com
          /foo          s1:80
          /bar          s2:80
```

只要服务（s1，s2）存在，Ingress controller就会将提供一个满足该Ingress的特定loadbalancer实现。 这一步完成后，您将在Ingress的最后一列看到loadbalancer的地址。

基于名称的虚拟主机

Name-based的虚拟主机在同一个IP地址下拥有多个主机名。

```
foo.bar.com --|                 |-> foo.bar.com s1:80
              | 178.91.123.132  |
bar.foo.com --|                 |-> bar.foo.com s2:80
```

下面这个ingress说明基于Host header的后端loadbalancer的路由请求：

```yaml
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: test
spec:
  rules:
  - host: foo.bar.com
    http:
      paths:
      - backend:
          serviceName: s1
          servicePort: 80
  - host: bar.foo.com
    http:
      paths:
      - backend:
          serviceName: s2
          servicePort: 80
```

默认**backend**：一个没有rule的ingress，如前面章节中所示，所有流量都将发送到一个默认backend。你可以用该技巧通知loadbalancer如何找到你网站的404页面，通过制定一些列rule和一个默认backend的方式。如果请求header中的host不能跟ingress中的host匹配，并且/或请求的URL不能与任何一个path匹配，则流量将路由到你的默认backend。

TLS

你可以通过指定包含TLS私钥和证书的secret来加密Ingress。 目前，Ingress仅支持单个TLS端口443，并假定TLS termination。 如果Ingress中的TLS配置部分指定了不同的主机，则它们将根据通过SNI TLS扩展指定的主机名（假如Ingress controller支持SNI）在多个相同端口上进行复用。 TLS secret中必须包含名为 `tls.crt` 和 `tls.key` 的密钥，这里面包含了用于TLS的证书和私钥，例如：

```yaml
apiVersion: v1
data:
  tls.crt: base64 encoded cert
  tls.key: base64 encoded key
kind: Secret
metadata:
  name: testsecret
  namespace: default
type: Opaque
```

在Ingress中引用这个secret将通知Ingress controller使用TLS加密从将客户端到loadbalancer的channel：

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: no-rules-map
spec:
  tls:
    - secretName: testsecret
  backend:
    serviceName: s1
    servicePort: 80
```

请注意，各种Ingress controller支持的TLS功能之间存在差距。 请参阅有关nginx，GCE或任何其他平台特定Ingress controller的文档，以了解TLS在你的环境中的工作原理。

Ingress controller启动时附带一些适用于所有Ingress的负载平衡策略设置，例如负载均衡算法，后端权重方案等。更高级的负载平衡概念（例如持久会话，动态权重）尚未在Ingress中公开。 你仍然可以通过service loadbalancer获取这些功能。随着时间的推移，我们计划将适用于跨平台的负载平衡模式加入到Ingress资源中。

还值得注意的是，尽管健康检查不直接通过Ingress公开，但Kubernetes中存在并行概念，例如准备探查，可以使你达成相同的最终结果。 请查看特定控制器的文档，以了解他们如何处理健康检查（nginx，GCE）。

更新Ingress

假如你想要向已有的ingress中增加一个新的Host，你可以编辑和更新该ingress：

```
$ kubectl get ing
NAME      RULE          BACKEND   ADDRESS
test      -                       178.91.123.132
          foo.bar.com
          /foo          s1:80
$ kubectl edit ing test
```

这会弹出一个包含已有的yaml文件的编辑器，修改它，增加新的Host配置。

```
spec:
  rules:
  - host: foo.bar.com
    http:
      paths:
      - backend:
          serviceName: s1
          servicePort: 80
        path: /foo
  - host: bar.baz.com
    http:
      paths:
      - backend:
          serviceName: s2
          servicePort: 80
        path: /foo
..
```

保存它会更新API server中的资源，这会触发ingress controller重新配置loadbalancer。

```
$ kubectl get ing
NAME      RULE          BACKEND   ADDRESS
test      -                       178.91.123.132
          foo.bar.com
          /foo          s1:80
          bar.baz.com
          /foo          s2:80
```

在一个修改过的ingress yaml文件上调用 `kubectl replace -f` 命令一样可以达到同样的效果。

跨可用域故障

在不通云供应商之间，跨故障域的流量传播技术有所不同。 有关详细信息，请查看相关Ingress controller的文档。 有关在federation集群中部署Ingress的详细信息，请参阅federation文档。

未来计划

- 多样化的HTTPS/TLS模型支持（如SNI，re-encryption）
- 通过声明来请求IP或者主机名

- 结合L4和L7 Ingress
- 更多的Ingress controller

请跟踪L7和Ingress的proposal，了解有关资源演进的更多细节，以及Ingress repository，了解有关各种Ingress controller演进的更多详细信息。

替代方案

你可以通过很多种方式暴露service而不必直接使用ingress：

- 使用Service.Type=LoadBalancer
- 使用Service.Type=NodePort
- 使用Port Proxy
- 部署一个Service loadbalancer 这允许你在多个service之间共享单个IP，并通过Service Annotations实现更高级的负载平衡。

参考

Kubernetes Ingress Resource

使用NGINX Plus负载均衡Kubernetes服务

使用 NGINX 和 NGINX Plus 的 Ingress Controller 进行 Kubernetes 的负载均衡

Kubernetes : Ingress Controller with Træfɪk and Let's Encrypt

Kubernetes : Træfɪk and Let's Encrypt at scale

Kubernetes Ingress Controller-Træfɪk

Kubernetes 1.2 and simplifying advanced networking with Ingress

Traefik Ingress Controller

我们在前面部署了 Traefik 作为Ingress Controller，如果集群外部直接访问Kubenretes内部服务的话，可以直接创建Ingress如下所示：

```yaml
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: traefik-ingress
  namespace: default
spec:
  rules:
  - host: traefik.nginx.io
    http:
      paths:
      - path: /
        backend:
          serviceName: my-nginx
          servicePort: 80
```

Traefik Ingress Controller

当我们处于迁移应用到kuberentes上的阶段时，可能有部分服务实例不在kubernetes上，服务的路由使用nginx配置，这时处于nginx和ingress共存的状态。参考下面的配置：

```yaml
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: td-ingress
  namespace: default
  annotations:
    traefik.frontend.rule.type: PathPrefixStrip
    kubernetes.io/ingress.class: traefik
spec:
  rules:
  - host: "*.jimmysong.io"
    http:
      paths:
      - path: /docGenerate
        backend:
          serviceName: td-sdmk-docgenerate
          servicePort: 80
```

注意**annotation**的配置：

- `traefik.frontend.rule.type: PathPrefixStrip`：表示将截掉URL中的 `path`
- `kubernetes.io/ingress.class`：表示使用的ingress类型

关于Ingress annotation的更多信息请参考：Ingress Annotations - kubernetes.io。

在nginx中增加配置：

```
upstream docGenerate {
       server 172.20.0.119:80;
       keepalive 200;
    }
```

172.20.0.119是我们的边缘节点的VIP，见边缘节点配置。

参考

- Kubernetes Ingress Backend - traefik.io
- Kubernetes Ingress Controller - traefik.io

身份与权限认证

Kubernetes中提供了良好的多租户认证管理机制，如RBAC、ServiceAccount还有各种Policy等。

Service Account

Service account为Pod中的进程提供身份信息。

本文是关于 *Service Account* 的用户指南，管理指南另见 *Service Account* 的集群管理指南 。

注意：本文档描述的关于 *Serivce Account* 的行为只有当您按照 *Kubernetes* 项目建议的方式搭建起集群的情况下才有效。您的集群管理员可能在您的集群中有自定义配置，这种情况下该文档可能并不适用。

当您（真人用户）访问集群（例如使用 `kubectl` 命令）时，apiserver 会将您认证为一个特定的 User Account（目前通常是 `admin` ，除非您的系统管理员自定义了集群配置）。Pod 容器中的进程也可以与 apiserver 联系。 当它们在联系 apiserver 的时候，它们会被认证为一个特定的 Service Account（例如 `default` ）。

使用默认的 Service Account 访问 API server

当您创建 pod 的时候，如果您没有指定一个 service account，系统会自动得在与该pod 相同的 namespace 下为其指派一个 `default` service account。如果您获取刚创建的 pod 的原始 json 或 yaml 信息（例如使用 `kubectl get pods/podename -o yaml` 命令），您将看到 `spec.serviceAccountName` 字段已经被设置为 automatically set 。

您可以在 pod 中使用自动挂载的 service account 凭证来访问 API，如 Accessing the Cluster 中所描述。

Service account 是否能够取得访问 API 的许可取决于您使用的 授权插件和策略。

在 1.6 以上版本中，您可以选择取消为 service account 自动挂载 API 凭证，只需在 service account 中设置 `automountServiceAccountToken: false` ：

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: build-robot
automountServiceAccountToken: false
...
```

在 1.6 以上版本中，您也可以选择只取消单个 pod 的 API 凭证自动挂载：

```
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  serviceAccountName: build-robot
  automountServiceAccountToken: false
  ...
```

如果在 pod 和 service account 中同时设置了 `automountServiceAccountToken` , pod 设置中的优先级更高。

使用多个Service Account

每个 namespace 中都有一个默认的叫做 `default` 的 service account 资源。

您可以使用以下命令列出 namespace 下的所有 serviceAccount 资源。

```
$ kubectl get serviceAccounts
NAME        SECRETS     AGE
default     1           1d
```

您可以像这样创建一个 ServiceAccount 对象：

```
$ cat > /tmp/serviceaccount.yaml <<EOF
apiVersion: v1
kind: ServiceAccount
metadata:
  name: build-robot
EOF
$ kubectl create -f /tmp/serviceaccount.yaml
serviceaccount "build-robot" created
```

如果您看到如下的 service account 对象的完整输出信息：

```
$ kubectl get serviceaccounts/build-robot -o yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  creationTimestamp: 2015-06-16T00:12:59Z
  name: build-robot
  namespace: default
  resourceVersion: "272500"
  selfLink: /api/v1/namespaces/default/serviceaccounts/build-rob
ot
  uid: 721ab723-13bc-11e5-aec2-42010af0021e
secrets:
- name: build-robot-token-bvbk5
```

然后您将看到有一个 token 已经被自动创建，并被 service account 引用。

您可以使用授权插件来 设置 service account 的权限 。

设置非默认的 service account，只需要在 pod 的 `spec.serviceAccountName` 字段中将name设置为您想要用的 service account 名字即可。

在 pod 创建之初 service account 就必须已经存在，否则创建将被拒绝。

您不能更新已创建的 pod 的 service account。

您可以清理 service account，如下所示：

```
$ kubectl delete serviceaccount/build-robot
```

手动创建 service account 的 API token

假设我们已经有了一个如上文提到的名为 ”build-robot“ 的 service account，我们手动创建一个新的 secret。

```
$ cat > /tmp/build-robot-secret.yaml <<EOF
apiVersion: v1
kind: Secret
metadata:
  name: build-robot-secret
  annotations:
    kubernetes.io/service-account.name: build-robot
type: kubernetes.io/service-account-token
EOF
$ kubectl create -f /tmp/build-robot-secret.yaml
secret "build-robot-secret" created
```

现在您可以确认下新创建的 secret 取代了 “build-robot” 这个 service account 原来的 API token。

所有已不存在的 service account 的 token 将被 token controller 清理掉。

```
$ kubectl describe secrets/build-robot-secret
Name:    build-robot-secret
Namespace:   default
Labels:    <none>
Annotations:   kubernetes.io/service-account.name=build-robot,kubernetes.io/service-account.uid=870ef2a5-35cf-11e5-8d06-005056b45392

Type: kubernetes.io/service-account-token

Data
====
ca.crt: 1220 bytes
token: ...
namespace: 7 bytes
```

> 注意该内容中的 `token` 被省略了。

为 service account 添加 ImagePullSecret

首先，创建一个 imagePullSecret，详见这里。

然后，确认已创建。如：

```
$ kubectl get secrets myregistrykey
NAME             TYPE                              DATA      AGE
myregistrykey    kubernetes.io/.dockerconfigjson   1         1d
```

然后，修改 namespace 中的默认 service account 使用该 secret 作为 imagePullSecret。

```
kubectl patch serviceaccount default -p '{"imagePullSecrets": [{
"name": "myregistrykey"}]}'
```

Vi 交互过程中需要手动编辑：

```
$ kubectl get serviceaccounts default -o yaml > ./sa.yaml
$ cat sa.yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  creationTimestamp: 2015-08-07T22:02:39Z
  name: default
  namespace: default
  resourceVersion: "243024"
  selfLink: /api/v1/namespaces/default/serviceaccounts/default
  uid: 052fb0f4-3d50-11e5-b066-42010af0d7b6
secrets:
- name: default-token-uudge
$ vi sa.yaml
[editor session not shown]
[delete line with key "resourceVersion"]
[add lines with "imagePullSecret:"]
$ cat sa.yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  creationTimestamp: 2015-08-07T22:02:39Z
  name: default
  namespace: default
  selfLink: /api/v1/namespaces/default/serviceaccounts/default
  uid: 052fb0f4-3d50-11e5-b066-42010af0d7b6
secrets:
- name: default-token-uudge
imagePullSecrets:
- name: myregistrykey
$ kubectl replace serviceaccount default -f ./sa.yaml
serviceaccounts/default
```

现在，所有当前 namespace 中新创建的 pod 的 spec 中都会增加如下内容：

```
spec:
  imagePullSecrets:
  - name: myregistrykey
```

参考

https://kubernetes.io/docs/tasks/configure-pod-container/configure-service-account/

RBAC——基于角色的访问控制

以下内容是 xingzhou 对 kubernetes 官方文档的翻译，原文地址 https://k8smeetup.github.io/docs/admin/authorization/rbac/

基于角色的访问控制（Role-Based Access Control, 即”RBAC”）使用”rbac.authorization.k8s.io” API Group实现授权决策，允许管理员通过Kubernetes API动态配置策略。

截至Kubernetes 1.6，RBAC模式处于beta版本。

要启用RBAC，请使用 `--authorization-mode=RBAC` 启动API Server。

API概述

本节将介绍RBAC API所定义的四种顶级类型。用户可以像使用其他Kubernetes API资源一样 （例如通过 `kubectl` 、API调用等）与这些资源进行交互。例如，命令 `kubectl create -f (resource).yml` 可以被用于以下所有的例子，当然，读者在尝试前可能需要先阅读以下相关章节的内容。

Role与ClusterRole

在RBAC API中，一个角色包含了一套表示一组权限的规则。 权限以纯粹的累加形式累积（没有”否定”的规则）。 角色可以由命名空间（namespace）内的 `Role` 对象定义，而整个Kubernetes集群范围内有效的角色则通过 `ClusterRole` 对象实现。

一个 `Role` 对象只能用于授予对某一单一命名空间中资源的访问权限。 以下示例描述了”default”命名空间中的一个 `Role` 对象的定义，用于授予对pod的读访问权限：

```
kind: Role
apiVersion: rbac.authorization.k8s.io/v1beta1
metadata:
  namespace: default
  name: pod-reader
rules:
- apiGroups: [""] # 空字符串""表明使用core API group
  resources: ["pods"]
  verbs: ["get", "watch", "list"]
```

`ClusterRole` 对象可以授予与 `Role` 对象相同的权限，但由于它们属于集群范围对象， 也可以使用它们授予对以下几种资源的访问权限：

- 集群范围资源（例如节点，即node）
- 非资源类型endpoint（例如”/healthz”）
- 跨所有命名空间的命名空间范围资源（例如pod，需要运行命令 `kubectl get pods --all-namespaces` 来查询集群中所有的pod）

下面示例中的 `ClusterRole` 定义可用于授予用户对某一特定命名空间，或者所有命名空间中的secret（取决于其绑定方式）的读访问权限：

```
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1beta1
metadata:
  # 鉴于ClusterRole是集群范围对象，所以这里不需要定义"namespace"字段
  name: secret-reader
rules:
- apiGroups: [""]
  resources: ["secrets"]
  verbs: ["get", "watch", "list"]
```

RoleBinding与ClusterRoleBinding

角色绑定将一个角色中定义的各种权限授予一个或者一组用户。 角色绑定包含了一组相关主体（即subject, 包括用户——User、用户组——Group、或者服务账户——Service Account）以及对被授予角色的引用。 在命名空间中可以通过 `RoleBinding` 对象授予权限，而集群范围的权限授予则通过 `ClusterRoleBinding` 对象完成。

`RoleBinding` 可以引用在同一命名空间内定义的 `Role` 对象。 下面示例中定义的 `RoleBinding` 对象在"default"命名空间中将"pod-reader"角色授予用户"jane"。这一授权将允许用户"jane"从"default"命名空间中读取pod。

```
# 以下角色绑定定义将允许用户"jane"从"default"命名空间中读取pod。
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1beta1
metadata:
  name: read-pods
  namespace: default
subjects:
- kind: User
  name: jane
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
  name: pod-reader
  apiGroup: rbac.authorization.k8s.io
```

`RoleBinding` 对象也可以引用一个 `ClusterRole` 对象用于在 `RoleBinding` 所在的命名空间内授予用户对所引用的 `ClusterRole` 中 定义的命名空间资源的访问权限。这一点允许管理员在整个集群范围内首先定义一组通用的角色，然后再在不同的命名空间中复用这些角色。

例如，尽管下面示例中的 `RoleBinding` 引用的是一个 `ClusterRole` 对象，但是用户"dave"（即角色绑定主体）还是只能读取"development" 命名空间中的secret（即 `RoleBinding` 所在的命名空间）。

```
# 以下角色绑定允许用户"dave"读取"development"命名空间中的secret。
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1beta1
metadata:
  name: read-secrets
  namespace: development # 这里表明仅授权读取"development"命名空间中
的资源。
subjects:
- kind: User
  name: dave
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: ClusterRole
  name: secret-reader
  apiGroup: rbac.authorization.k8s.io
```

最后，可以使用 `ClusterRoleBinding` 在集群级别和所有命名空间中授予权限。下面示例中所定义的 `ClusterRoleBinding` 允许在用户组"manager"中的任何用户都可以读取集群中任何命名空间中的secret。

```
# 以下`ClusterRoleBinding`对象允许在用户组"manager"中的任何用户都可以读取集群中任何命名空间中的secret。
kind: ClusterRoleBinding
apiVersion: rbac.authorization.k8s.io/v1beta1
metadata:
  name: read-secrets-global
subjects:
- kind: Group
  name: manager
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: ClusterRole
  name: secret-reader
  apiGroup: rbac.authorization.k8s.io
```

对资源的引用

大多数资源由代表其名字的字符串表示，例如"pods"，就像它们出现在相关API endpoint的URL中一样。然而，有一些Kubernetes API还 包含了"子资源"，比如pod的logs。在Kubernetes中，pod logs endpoint的URL格式为：

```
GET /api/v1/namespaces/{namespace}/pods/{name}/log
```

在这种情况下，"pods"是命名空间资源，而"log"是pods的子资源。为了在RBAC角色中表示出这一点，我们需要使用斜线来划分资源 与子资源。如果需要角色绑定主体读取pods以及pod log，您需要定义以下角色：

```
kind: Role
apiVersion: rbac.authorization.k8s.io/v1beta1
metadata:
  namespace: default
  name: pod-and-pod-logs-reader
rules:
- apiGroups: [""]
  resources: ["pods", "pods/log"]
  verbs: ["get", "list"]
```

通过 `resourceNames` 列表，角色可以针对不同种类的请求根据资源名引用资源实例。当指定了 `resourceNames` 列表时，不同动作 种类的请求的权限，如使用”get”、”delete”、”update”以及”patch”等动词的请求，将被限定到资源列表中所包含的资源实例上。 例如，如果需要限定一个角色绑定主体只能”get”或者”update”一个configmap时，您可以定义以下角色：

```
kind: Role
apiVersion: rbac.authorization.k8s.io/v1beta1
metadata:
  namespace: default
  name: configmap-updater
rules:
- apiGroups: [""]
  resources: ["configmap"]
  resourceNames: ["my-configmap"]
  verbs: ["update", "get"]
```

值得注意的是，如果设置了 `resourceNames` ，则请求所使用的动词不能是list、watch、create或者deletecollection。 由于资源名不会出现在create、list、watch和deletecollection等API请求的URL中，所以这些请求动词不会被设置了 `resourceNames` 的规则所允许，因为规则中的 `resourceNames` 部分不会匹配这些请求。

一些角色定义的例子

在以下示例中，我们仅截取展示了 `rules` 部分的定义。

允许读取core API Group中定义的资源”pods”：

```
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["get", "list", "watch"]
```

允许读写在”extensions”和”apps” API Group中定义的”deployments”：

```
rules:
- apiGroups: ["extensions", "apps"]
  resources: ["deployments"]
  verbs: ["get", "list", "watch", "create", "update", "patch", "
delete"]
```

允许读取"pods"以及读写"jobs"：

```
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["get", "list", "watch"]
- apiGroups: ["batch", "extensions"]
  resources: ["jobs"]
  verbs: ["get", "list", "watch", "create", "update", "patch", "
delete"]
```

允许读取一个名为"my-config"的 `ConfigMap` 实例（需要将其通过 `RoleBinding` 绑定从而限制针对某一个命名空间中定义的一个 `ConfigMap` 实例的访问）：

```
rules:
- apiGroups: [""]
  resources: ["configmaps"]
  resourceNames: ["my-config"]
  verbs: ["get"]
```

允许读取core API Group中的"nodes"资源（由于 `Node` 是集群级别资源，所以此 `ClusterRole` 定义需要与一个 `ClusterRoleBinding` 绑定才能有效）：

```
rules:
- apiGroups: [""]
  resources: ["nodes"]
  verbs: ["get", "list", "watch"]
```

允许对非资源endpoint "/healthz"及其所有子路径的"GET"和"POST"请求（此 `ClusterRole` 定义需要与一个 `ClusterRoleBinding` 绑定才能有效）：

```
rules:
- nonResourceURLs: ["/healthz", "/healthz/*"] # 在非资源URL中，'*'
代表后缀通配符
  verbs: ["get", "post"]
```

对角色绑定主体（Subject）的引用

`RoleBinding` 或者 `ClusterRoleBinding` 将角色绑定到角色绑定主体（Subject）。角色绑定主体可以是用户组（Group）、用户（User）或者服务账户（Service Accounts）。

用户由字符串表示。可以是纯粹的用户名，例如"alice"、电子邮件风格的名字，如 "bob@example.com" 或者是用字符串表示的数字id。由Kubernetes管理员配置认证模块 以产生所需格式的用户名。对于用户名，RBAC授权系统不要求任何特定的格式。然而，前缀 `system:` 是 为Kubernetes系统使用而保留的，所以管理员应该确保用户名不会意外地包含这个前缀。

Kubernetes中的用户组信息由授权模块提供。用户组与用户一样由字符串表示。Kubernetes对用户组 字符串没有格式要求，但前缀 `system:` 同样是被系统保留的。

服务账户拥有包含 `system:serviceaccount:` 前缀的用户名，并属于拥有 `system:serviceaccounts:` 前缀的用户组。

角色绑定的一些例子

以下示例中，仅截取展示了 `RoleBinding` 的 `subjects` 字段。

一个名为"alice@example.com"的用户：

```yaml
subjects:
- kind: User
  name: "alice@example.com"
  apiGroup: rbac.authorization.k8s.io
```

一个名为"frontend-admins"的用户组：

```yaml
subjects:
- kind: Group
  name: "frontend-admins"
  apiGroup: rbac.authorization.k8s.io
```

kube-system命名空间中的默认服务账户：

```yaml
subjects:
- kind: ServiceAccount
  name: default
  namespace: kube-system
```

名为"qa"命名空间中的所有服务账户：

```yaml
subjects:
- kind: Group
  name: system:serviceaccounts:qa
  apiGroup: rbac.authorization.k8s.io
```

在集群中的所有服务账户：

```yaml
subjects:
- kind: Group
  name: system:serviceaccounts
  apiGroup: rbac.authorization.k8s.io
```

所有认证过的用户（version 1.5+）：

```yaml
subjects:
- kind: Group
  name: system:authenticated
  apiGroup: rbac.authorization.k8s.io
```

所有未认证的用户（version 1.5+）：

```yaml
subjects:
- kind: Group
  name: system:unauthenticated
  apiGroup: rbac.authorization.k8s.io
```

所有用户（version 1.5+）：

```yaml
subjects:
- kind: Group
  name: system:authenticated
  apiGroup: rbac.authorization.k8s.io
- kind: Group
  name: system:unauthenticated
  apiGroup: rbac.authorization.k8s.io
```

默认角色与默认角色绑定

API Server会创建一组默认的 `ClusterRole` 和 `ClusterRoleBinding` 对象。 这些默认对象中有许多包含 `system:` 前缀，表明这些资源由Kubernetes基础组件"拥有"。 对这些资源的修改可能导致非功能性集群（non-functional cluster）。一个例子是 `system:node` ClusterRole对象。 这个角色定义了kubelets的权限。如果这个角色被修改，可能会导致kubelets无法正常工作。

所有默认的ClusterRole和ClusterRoleBinding对象都会被标记为 `kubernetes.io/bootstrapping=rbac-defaults` 。

自动更新

每次启动时，API Server都会更新默认ClusterRole所缺乏的各种权限，并更新默认ClusterRoleBinding所缺乏的各个角色绑定主体。 这种自动更新机制允许集群修复一些意外的修改。由于权限和角色绑定主体在新的Kubernetes释出版本中可能变化，这也能够保证角色和角色 绑定始终保持是最新的。

如果需要禁用自动更新，请将默认ClusterRole以及ClusterRoleBinding的 `rbac.authorization.kubernetes.io/autoupdate` 设置成为 `false` 。 请注意，缺乏默认权限和角色绑定主体可能会导致非功能性集群问题。

自Kubernetes 1.6+起，当集群RBAC授权器（RBAC Authorizer）处于开启状态时，可以启用自动更新功能.

发现类角色

默认ClusterRole	默认ClusterRoleBinding	描述
system:basic-user	**system:authenticated** and **system:unauthenticated**groups	允许用户只读访问有关自己的基本信息。
system:discovery	**system:authenticated** and **system:unauthenticated**groups	允许只读访问API discovery endpoints, 用于在API级别进行发现和协商。

面向用户的角色

一些默认角色并不包含 `system:` 前缀，它们是面向用户的角色。 这些角色包含超级用户角色（ `cluster-admin` ），即旨在利用ClusterRoleBinding（ `cluster-status` ）在集群范围内授权的角色， 以及那些使用
RoleBinding（ `admin` 、 `edit` 和 `view` ）在特定命名空间中授权的角色。

默认 ClusterRole	默认 ClusterRoleBinding	描述
cluster-admin	**system:masters** group	超级用户权限，允许对任何资源执行任何操作。 在**ClusterRoleBinding**中使用时，可以完全控制集群和所有命名空间中的所有资源。 在**RoleBinding**中使用时，可以完全控制RoleBinding所在命名空间中的所有资源，包括命名空间自己。
admin	None	管理员权限，利用**RoleBinding**在某一命名空间内部授予。 在**RoleBinding**中使用时，允许针对命名空间内大部分资源的读写访问， 包括在命名空间内创建角色与角色绑定的能力。 但不允许对资源配额（resource quota）或者命名空间本身的写访问。
edit	None	允许对某一个命名空间内大部分对象的读写访问，但不允许查看或者修改角色或者角色绑定。
view	None	允许对某一个命名空间内大部分对象的只读访问。 不允许查看角色或者角色绑定。 由于可扩散性等原因，不允许查看secret资源。

Core Component Roles

核心组件角色

默认 ClusterRole	默认 ClusterRoleBinding	描述
system:kube-scheduler	**system:kube-scheduler** user	允许访问kube-scheduler组件所需要的资源。
system:kube-controller-manager	**system:kube-controller-manager** user	允许访问kube-controller-manager组件所需要的资源。 单个控制循环所需要的权限请参阅控制器（controller）角色.
system:node	**system:nodes** group (deprecated in 1.7)	允许对kubelet组件所需要的资源的访问，包括读取所有**secret**和对所有**pod**的写访问。 自Kubernetes 1.7开始, 相比较于这个角色，更推荐使用Node authorizer 以及 NodeRestriction admission plugin，并允许根据调度运行在节点上的pod授予kubelets API访问的权限。 自Kubernetes 1.7开始，当启用 `Node` 授权模式时，对 `system:nodes` 用户组的绑定将不会被自动创建。
system:node-proxier	**system:kube-proxy** user	允许对kube-proxy组件所需要资源的访问。

其它组件角色

默认ClusterRole	默认ClusterRoleBinding	描述
system:auth-delegator	None	允许委托认证和授权检查。 通常由附加API Server用于统一认证和授权。
system:heapster	None	Heapster组件的角色。
system:kube-aggregator	None	kube-aggregator组件的角色。
system:kube-dns	**kube-dns** service account in the **kube-system**namespace	kube-dns组件的角色。
system:node-bootstrapper	None	允许对执行Kubelet TLS引导（Kubelet TLS bootstrapping）所需要资源的访问.
system:node-problem-detector	None	node-problem-detector组件的角色。
system:persistent-volume-provisioner	None	允许对大部分动态存储卷创建组件（dynamic volume provisioner）所需要资源的访问。

控制器（Controller）角色

Kubernetes controller manager负责运行核心控制循环。 当使用 `--use-service-account-credentials` 选项运行controller manager时，每个控制循环都将使用单独的服务账户启动。 而每个控制循环都存在对应的角色，前缀名为 `system:controller:` 。 如果不使用 `--use-service-account-credentials` 选项时，controller manager将会使用自己的凭证运行所有控制循环，而这些凭证必须被授予相关的角色。 这些角色包括：

- system:controller:attachdetach-controller
- system:controller:certificate-controller
- system:controller:cronjob-controller
- system:controller:daemon-set-controller
- system:controller:deployment-controller
- system:controller:disruption-controller
- system:controller:endpoint-controller

- system:controller:generic-garbage-collector
- system:controller:horizontal-pod-autoscaler
- system:controller:job-controller
- system:controller:namespace-controller
- system:controller:node-controller
- system:controller:persistent-volume-binder
- system:controller:pod-garbage-collector
- system:controller:replicaset-controller
- system:controller:replication-controller
- system:controller:resourcequota-controller
- system:controller:route-controller
- system:controller:service-account-controller
- system:controller:service-controller
- system:controller:statefulset-controller
- system:controller:ttl-controller

初始化与预防权限升级

RBAC API会阻止用户通过编辑角色或者角色绑定来升级权限。 由于这一点是在API级别实现的，所以在RBAC授权器（RBAC authorizer）未启用的状态下依然可以正常工作。

用户只有在拥有了角色所包含的所有权限的条件下才能创建／更新一个角色，这些操作还必须在角色所处的相同范围内进行（对于 `ClusterRole` 来说是集群范围，对于 `Role` 来说是在与角色相同的命名空间或者集群范围）。 例如，如果用户"user-1"没有权限读取集群范围内的secret列表，那么他也不能创建包含这种权限的 `ClusterRole` 。为了能够让用户创建／更新角色，需要：

1. 授予用户一个角色以允许他们根据需要创建／更新 `Role` 或者 `ClusterRole` 对象。
2. 授予用户一个角色包含他们在 `Role` 或者 `ClusterRole` 中所能够设置的所有权限。如果用户尝试创建或者修改 `Role` 或者 `ClusterRole` 以设置那些他们未被授权的权限时，这些API请求将被禁止。

用户只有在拥有所引用的角色中包含的所有权限时才可以创建／更新角色绑定（这些操作也必须在角色绑定所处的相同范围内进行）或者用户被明确授权可以在所引用的角色上执行绑定操作。 例如，如果用户"user-1"没有权限读取集群范围内的

secret列表，那么他将不能创建 `ClusterRole` 来引用那些授予了此项权限的角色。为了能够让用户创建／更新角色绑定，需要：

1. 授予用户一个角色以允许他们根据需要创建／更新 `RoleBinding` 或者 `ClusterRoleBinding` 对象。
2. 授予用户绑定某一特定角色所需要的权限：
    - 隐式地，通过授予用户所有所引用的角色中所包含的权限
    - 显式地，通过授予用户在特定Role（或者ClusterRole）对象上执行 `bind` 操作的权限

例如，下面例子中的ClusterRole和RoleBinding将允许用户"user-1"授予其它用户"user-1-namespace"命名空间内的 `admin` 、 `edit` 和 `view` 等角色和角色绑定。

```
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: ClusterRole
metadata:
  name: role-grantor
rules:
- apiGroups: ["rbac.authorization.k8s.io"]
  resources: ["rolebindings"]
  verbs: ["create"]
- apiGroups: ["rbac.authorization.k8s.io"]
  resources: ["clusterroles"]
  verbs: ["bind"]
  resourceNames: ["admin","edit","view"]
---
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: RoleBinding
metadata:
  name: role-grantor-binding
  namespace: user-1-namespace
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: role-grantor
subjects:
- apiGroup: rbac.authorization.k8s.io
  kind: User
  name: user-1
```

当初始化第一个角色和角色绑定时，初始用户需要能够授予他们尚未拥有的权限。初始化初始角色和角色绑定时需要：

- 使用包含 `system：masters` 用户组的凭证，该用户组通过默认绑定绑定

到 `cluster-admin` 超级用户角色。
- 如果您的API Server在运行时启用了非安全端口（ `--insecure-port` ），您也可以通过这个没有施行认证或者授权的端口发送角色或者角色绑定请求。

一些命令行工具

有两个 `kubectl` 命令可以用于在命名空间内或者整个集群内授予角色。

`kubectl create rolebinding`

在某一特定命名空间内授予 `Role` 或者 `ClusterRole` 。示例如下：

- 在名为”acme”的命名空间中将 `admin` `ClusterRole` 授予用户”bob”：

  ```
  kubectl create rolebinding bob-admin-binding --clusterrole=admin --user=bob --namespace=acme
  ```

- 在名为”acme”的命名空间中将 `view` `ClusterRole` 授予服务账户”myapp”：

  ```
  kubectl create rolebinding myapp-view-binding --clusterrole=view --serviceaccount=acme:myapp --namespace=acme
  ```

`kubectl create clusterrolebinding`

在整个集群中授予 `ClusterRole` ，包括所有命名空间。示例如下：

- 在整个集群范围内将 `cluster-admin` `ClusterRole` 授予用户”root”：

  ```
  kubectl create clusterrolebinding root-cluster-admin-binding --clusterrole=cluster-admin --user=root
  ```

- 在整个集群范围内将 `system:node` `ClusterRole` 授予用户”kubelet”：

  ```
  kubectl create clusterrolebinding kubelet-node-binding --clusterrole=system:node --user=kubelet
  ```

- 在整个集群范围内将 `view` `ClusterRole` 授予命名空间”acme”内的服务账户”myapp”：

  ```
  kubectl create clusterrolebinding myapp-view-binding --clusterrole=view --serviceaccount=acme:myapp
  ```

请参阅CLI帮助文档以获得上述命令的详细用法

服务账户（Service Account）权限

默认的RBAC策略将授予控制平面组件（control-plane component）、节点（node）和控制器（controller）一组范围受限的权限，但对于”kube-system”命名空间以外的服务账户，则不授予任何权限（超出授予所有认证用户的发现权限）。

这一点允许您根据需要向特定服务账号授予特定权限。细粒度的角色绑定将提供更好的安全性，但需要更多精力管理。更粗粒度的授权可能授予服务账号不需要的API访问权限（甚至导致潜在授权扩散），但更易于管理。

从最安全到最不安全可以排序以下方法：

1. 对某一特定应用程序的服务账户授予角色（最佳实践）

   要求应用程序在其pod规范（pod spec）中指定 `serviceAccountName` 字段，并且要创建相应服务账户（例如通过API、应用程序清单或者命令 `kubectl create serviceaccount` 等）。

   例如，在”my-namespace”命名空间中授予服务账户”my-sa”只读权限：

   ```
   kubectl create rolebinding my-sa-view \
     --clusterrole=view \
     --serviceaccount=my-namespace:my-sa \
     --namespace=my-namespace
   ```

2. 在某一命名空间中授予”default”服务账号一个角色

   如果一个应用程序没有在其pod规范中指定 `serviceAccountName` ，它将默认使用”default”服务账号。

   注意：授予”default”服务账号的权限将可用于命名空间内任何没有指定 `serviceAccountName` 的pod。

   下面的例子将在”my-namespace”命名空间内授予”default”服务账号只读权限：

   ```
   kubectl create rolebinding default-view \
     --clusterrole=view \
     --serviceaccount=my-namespace:default \
     --namespace=my-namespace
   ```

目前，许多加载项（addon）作为”kube-system”命名空间中的”default”服务帐户运行。 要允许这些加载项使用超级用户访问权限，请将cluster-admin权限授予”kube-system”命名空间中的”default”服务帐户。 注意：启用上述操作意味着”kube-system”命名空间将包含允许超级用户访问API的秘钥。

```
kubectl create clusterrolebinding add-on-cluster-admin \
  --clusterrole=cluster-admin \
  --serviceaccount=kube-system:default
```

3. 为命名空间中所有的服务账号授予角色

   如果您希望命名空间内的所有应用程序都拥有同一个角色，无论它们使用什么服务账户，您可以为该命名空间的服务账户用户组授予角色。

   下面的例子将授予”my-namespace”命名空间中的所有服务账户只读权限：

```
kubectl create rolebinding serviceaccounts-view \
  --clusterrole=view \
  --group=system:serviceaccounts:my-namespace \
  --namespace=my-namespace
```

4. 对集群范围内的所有服务账户授予一个受限角色（不鼓励）

   如果您不想管理每个命名空间的权限，则可以将集群范围角色授予所有服务帐户。

   下面的例子将所有命名空间中的只读权限授予集群中的所有服务账户：

```
kubectl create clusterrolebinding serviceaccounts-view \
  --clusterrole=view \
  --group=system:serviceaccounts
```

5. 授予超级用户访问权限给集群范围内的所有服务帐户（强烈不鼓励）

   如果您根本不关心权限分块，您可以对所有服务账户授予超级用户访问权限。

   警告：这种做法将允许任何具有读取权限的用户访问secret或者通过创建一个容器的方式来访问超级用户的凭据。

```
kubectl create clusterrolebinding serviceaccounts-cluster-ad
min \
  --clusterrole=cluster-admin \
  --group=system:serviceaccounts
```

从版本1.5升级

在Kubernetes 1.6之前，许多部署使用非常宽泛的ABAC策略，包括授予对所有服务帐户的完整API访问权限。

默认的RBAC策略将授予控制平面组件（control-plane components）、节点（nodes）和控制器（controller）一组范围受限的权限， 但对于"kube-system"命名空间以外的服务账户，则不授予任何权限（超出授予所有认证用户的发现权限）。

虽然安全性更高，但这可能会影响到期望自动接收API权限的现有工作负载。 以下是管理此转换的两种方法：

并行授权器（authorizer）

同时运行RBAC和ABAC授权器，并包括旧版ABAC策略：

```
--authorization-mode=RBAC,ABAC --authorization-policy-file=mypol
icy.jsonl
```

RBAC授权器将尝试首先授权请求。如果RBAC授权器拒绝API请求，则ABAC授权器将被运行。这意味着RBAC策略或者ABAC策略所允许的任何请求都是可通过的。

当以日志级别为2或更高（ `--v = 2` ）运行时，您可以在API Server日志中看到RBAC拒绝请求信息（以 `RBAC DENY:` 为前缀）。 您可以使用该信息来确定哪些角色需要授予哪些用户，用户组或服务帐户。 一旦授予服务帐户角色，并且服务器日志中没有RBAC拒绝消息的工作负载正在运行，您可以删除ABAC授权器。

宽泛的RBAC权限

您可以使用RBAC角色绑定来复制一个宽泛的策略。

警告：以下政策略允许所有服务帐户作为集群管理员。 运行在容器中的任何应用程序都会自动接收服务帐户凭据，并且可以对**API**执行任何操作，包括查看**secret**和修改权限。 因此，并不推荐使用这种策略。

```
kubectl create clusterrolebinding permissive-binding \
  --clusterrole=cluster-admin \
  --user=admin \
  --user=kubelet \
  --group=system:serviceaccounts
```

Network Policy

网络策略说明一组 `Pod` 之间是如何被允许互相通信，以及如何与其它网络 Endpoint 进行通信。 `NetworkPolicy` 资源使用标签来选择 `Pod` ，并定义了一些规则，这些规则指明允许什么流量进入到选中的 `Pod` 上。

前提条件

网络策略通过网络插件来实现，所以必须使用一种支持 `NetworkPolicy` 的网络方案 —— 非 Controller 创建的资源，是不起作用的。

隔离的与未隔离的 Pod

默认 Pod 是未隔离的，它们可以从任何的源接收请求。 具有一个可以选择 Pod 的网络策略后，Pod 就会变成隔离的。 一旦 Namespace 中配置的网络策略能够选择一个特定的 Pod，这个 Pod 将拒绝任何该网络策略不允许的连接。（Namespace 中其它未被网络策略选中的 Pod 将继续接收所有流量）

`NetworkPolicy` 资源

查看 API参考 可以获取该资源的完整定义。

下面是一个 `NetworkPolicy` 的例子：

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: test-network-policy
  namespace: default
spec:
  podSelector:
    matchLabels:
      role: db
  ingress:
  - from:
    - namespaceSelector:
        matchLabels:
          project: myproject
    - podSelector:
        matchLabels:
          role: frontend
    ports:
    - protocol: TCP
      port: 6379
```

将上面配置 *POST* 到 *API Server* 将不起任何作用，除非选择的网络方案支持网络策略。

必选字段：像所有其它 Kubernetes 配置一样， `NetworkPolicy` 需要 `apiVersion` 、 `kind` 和 `metadata` 这三个字段，关于如何使用配置文件的基本信息，可以查看 这里，这里 和 这里 。

spec： `NetworkPolicy` spec 具有在给定 Namespace 中定义特定网络的全部信息。

podSelector：每个 `NetworkPolicy` 包含一个 `podSelector` ，它可以选择一组应用了网络策略的 Pod。由于 `NetworkPolicy` 当前只支持定义 `ingress` 规则，这个 `podSelector` 实际上为该策略定义了一组 “目标Pod”。示例中的策略选择了标签为 “role=db” 的 Pod。一个空的 `podSelector` 选择了该 Namespace 中的所有 Pod。

ingress：每个 `NetworkPolicy` 包含了一个白名单 `ingress` 规则列表。每个规则只允许能够匹配上 `from` 和 `ports` 配置段的流量。示例策略包含了单个规则，它从这两个源中匹配在单个端口上的流量，第一个是通过 `namespaceSelector` 指定的，第二个是通过 `podSelector` 指定的。

因此，上面示例的 NetworkPolicy：

1. 在 “default” Namespace中 隔离了标签 “role=db” 的 Pod（如果他们还没有被隔离）
2. 在 “default” Namespace中，允许任何具有 “role=frontend” 的 Pod，连接到标签为 “role=db” 的 Pod 的 TCP 端口 6379
3. 允许在 Namespace 中任何具有标签 “project=myproject” 的 Pod，连接到 “default” Namespace 中标签为 “role=db” 的 Pod 的 TCP 端口 6379

查看 NetworkPolicy 入门指南 给出的更进一步的例子。

默认策略

通过创建一个可以选择所有 Pod 但不允许任何流量的 NetworkPolicy，你可以为一个 Namespace 创建一个 “默认的” 隔离策略，如下所示：

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: default-deny
spec:
  podSelector:
```

这确保了即使是没有被任何 NetworkPolicy 选中的 Pod，将仍然是被隔离的。

可选地，在 Namespace 中，如果你想允许所有的流量进入到所有的 Pod（即使已经添加了某些策略，使一些 Pod 被处理为 “隔离的”），你可以通过创建一个策略来显式地指定允许所有流量：

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-all
spec:
  podSelector:
  ingress:
  - {}
```

原文地址：https://k8smeetup.github.io/docs/concepts/services-networking/network-policies/

译者：shirdrn

存储

为了管理存储，Kubernetes提供了Secret用于管理敏感信息，ConfigMap存储配置，Volume、PV、PVC、StorageClass等用来管理存储卷。

Secret

Secret解决了密码、token、密钥等敏感数据的配置问题，而不需要把这些敏感数据暴露到镜像或者Pod Spec中。Secret可以以Volume或者环境变量的方式使用。

Secret有三种类型：

- **Service Account** ：用来访问Kubernetes API，由Kubernetes自动创建，并且会自动挂载到Pod的 `/run/secrets/kubernetes.io/serviceaccount` 目录中；
- **Opaque** ：base64编码格式的Secret，用来存储密码、密钥等；
- **kubernetes.io/dockerconfigjson** ：用来存储私有docker registry的认证信息。

Opaque Secret

Opaque类型的数据是一个map类型，要求value是base64编码格式：

```
$ echo -n "admin" | base64
YWRtaW4=
$ echo -n "1f2d1e2e67df" | base64
MWYyZDFlMmU2N2Rm
```

secrets.yml

```
apiVersion: v1
kind: Secret
metadata:
  name: mysecret
type: Opaque
data:
  password: MWYyZDFlMmU2N2Rm
  username: YWRtaW4=
```

接着，就可以创建secret了： `kubectl create -f secrets.yml` 。

创建好secret之后，有两种方式来使用它：

- 以Volume方式

- 以环境变量方式

将Secret挂载到Volume中

```yaml
apiVersion: v1
kind: Pod
metadata:
  labels:
    name: db
  name: db
spec:
  volumes:
  - name: secrets
    secret:
      secretName: mysecret
  containers:
  - image: gcr.io/my_project_id/pg:v1
    name: db
    volumeMounts:
    - name: secrets
      mountPath: "/etc/secrets"
      readOnly: true
    ports:
    - name: cp
      containerPort: 5432
      hostPort: 5432
```

将Secret导出到环境变量中

```yaml
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: wordpress-deployment
spec:
  replicas: 2
  strategy:
      type: RollingUpdate
  template:
    metadata:
      labels:
        app: wordpress
        visualize: "true"
    spec:
      containers:
      - name: "wordpress"
        image: "wordpress"
        ports:
        - containerPort: 80
        env:
        - name: WORDPRESS_DB_USER
          valueFrom:
            secretKeyRef:
              name: mysecret
              key: username
        - name: WORDPRESS_DB_PASSWORD
          valueFrom:
            secretKeyRef:
              name: mysecret
              key: password
```

kubernetes.io/dockerconfigjson

可以直接用 `kubectl` 命令来创建用于docker registry认证的secret：

```
$ kubectl create secret docker-registry myregistrykey --docker-s
erver=DOCKER_REGISTRY_SERVER --docker-username=DOCKER_USER --doc
ker-password=DOCKER_PASSWORD --docker-email=DOCKER_EMAIL
secret "myregistrykey" created.
```

也可以直接读取 `~/.docker/config.json` 的内容来创建：

```
$ cat ~/.docker/config.json | base64
$ cat > myregistrykey.yaml <<EOF
apiVersion: v1
kind: Secret
metadata:
  name: myregistrykey
data:
  .dockerconfigjson: UmVhbGx5IHJlYWxseSByZWVlZWVlZWVlZWFhYWFhYWF
hYWFhYWFhYWFhYWFhYWFhYWFhYWxsbGxsbGxsbGxsbGxsbGxsbGxsbGxsbGx
sbGx5eXl5eXl5eXl5eXl5eXl5eSBsbGxsbGxsbGxsbGxsbG9vb29vb29vb29
vb29vb29vb29vb29vb29vb25ubm5ubm5ubm5ubm5ubm5ubm5ubmdnZ2dnZ2d
nZ2dnZ2dnZ2dnZ2cgYXV0aCBrZXlzCg==
type: kubernetes.io/dockerconfigjson
EOF
$ kubectl create -f myregistrykey.yaml
```

在创建Pod的时候，通过 `imagePullSecrets` 来引用刚创建的 `myregistrykey` :

```
apiVersion: v1
kind: Pod
metadata:
  name: foo
spec:
  containers:
    - name: foo
      image: janedoe/awesomeapp:v1
  imagePullSecrets:
    - name: myregistrykey
```

Service Account

Service Account用来访问Kubernetes API，由Kubernetes自动创建，并且会自动挂载到Pod的 `/run/secrets/kubernetes.io/serviceaccount` 目录中。

```
$ kubectl run nginx --image nginx
deployment "nginx" created
$ kubectl get pods
NAME                     READY     STATUS    RESTARTS   AGE
nginx-3137573019-md1u2   1/1       Running   0          13s
$ kubectl exec nginx-3137573019-md1u2 ls /run/secrets/kubernetes
.io/serviceaccount
ca.crt
namespace
token
```

ConfigMap

其实ConfigMap功能在Kubernetes1.2版本的时候就有了，许多应用程序会从配置文件、命令行参数或环境变量中读取配置信息。这些配置信息需要与docker image解耦，你总不能每修改一个配置就重做一个image吧？ConfigMap API给我们提供了向容器中注入配置信息的机制，ConfigMap可以被用来保存单个属性，也可以用来保存整个配置文件或者JSON二进制大对象。

ConfigMap概览

ConfigMap API资源用来保存**key-value pair**配置数据，这个数据可以在**pods**里使用，或者被用来为像**controller**一样的系统组件存储配置数据。虽然ConfigMap跟Secrets类似，但是ConfigMap更方便的处理不含敏感信息的字符串。 注意：ConfigMaps不是属性配置文件的替代品。ConfigMaps只是作为多个properties文件的引用。你可以把它理解为Linux系统中的 `/etc` 目录，专门用来存储配置文件的目录。下面举个例子，使用ConfigMap配置来创建Kuberntes Volumes，ConfigMap中的每个data项都会成为一个新文件。

```
kind: ConfigMap
apiVersion: v1
metadata:
  creationTimestamp: 2016-02-18T19:14:38Z
  name: example-config
  namespace: default
data:
  example.property.1: hello
  example.property.2: world
  example.property.file: |-
    property.1=value-1
    property.2=value-2
    property.3=value-3
```

`data` 一栏包括了配置数据，ConfigMap可以被用来保存单个属性，也可以用来保存一个配置文件。 配置数据可以通过很多种方式在Pods里被使用。ConfigMaps可以被用来：

1. 设置环境变量的值
2. 在容器里设置命令行参数

3. 在数据卷里面创建config文件

用户和系统组件两者都可以在ConfigMap里面存储配置数据。

其实不用看下面的文章，直接从 `kubectl create configmap -h` 的帮助信息中就可以对ConfigMap究竟如何创建略知一二了。

```
Examples:
  # Create a new configmap named my-config based on folder bar
  kubectl create configmap my-config --from-file=path/to/bar

  # Create a new configmap named my-config with specified keys instead of file basenames on disk
  kubectl create configmap my-config --from-file=key1=/path/to/bar/file1.txt --from-file=key2=/path/to/bar/file2.txt

  # Create a new configmap named my-config with key1=config1 and key2=config2
  kubectl create configmap my-config --from-literal=key1=config1 --from-literal=key2=config2
```

创建ConfigMaps

可以使用该命令，用给定值、文件或目录来创建ConfigMap。

```
kubectl create configmap
```

使用目录创建

比如我们已经有个了包含一些配置文件，其中包含了我们想要设置的ConfigMap的值：

```
$ ls docs/user-guide/configmap/kubectl/
game.properties
ui.properties

$ cat docs/user-guide/configmap/kubectl/game.properties
enemies=aliens
lives=3
enemies.cheat=true
enemies.cheat.level=noGoodRotten
secret.code.passphrase=UUDDLRLRBABAS
secret.code.allowed=true
secret.code.lives=30

$ cat docs/user-guide/configmap/kubectl/ui.properties
color.good=purple
color.bad=yellow
allow.textmode=true
how.nice.to.look=fairlyNice
```

使用下面的命令可以创建一个包含目录中所有文件的ConfigMap。

```
$ kubectl create configmap game-config --from-file=docs/user-gui
de/configmap/kubectl
```

`—from-file` 指定在目录下的所有文件都会被用在ConfigMap里面创建一个键值对，键的名字就是文件名，值就是文件的内容。

让我们来看一下这个命令创建的ConfigMap：

```
$ kubectl describe configmaps game-config
Name:           game-config
Namespace:      default
Labels:         <none>
Annotations:    <none>

Data
====
game.properties:        158 bytes
ui.properties:          83 bytes
```

我们可以看到那两个key是从kubectl指定的目录中的文件名。这些key的内容可能会很大，所以在kubectl describe的输出中，只能够看到键的名字和他们的大小。 如果想要看到键的值的话，可以使用 `kubectl get` ：

```
$ kubectl get configmaps game-config -o yaml
```

我们以 `yaml` 格式输出配置。

```yaml
apiVersion: v1
data:
  game.properties: |
    enemies=aliens
    lives=3
    enemies.cheat=true
    enemies.cheat.level=noGoodRotten
    secret.code.passphrase=UUDDLRLRBABAS
    secret.code.allowed=true
    secret.code.lives=30
  ui.properties: |
    color.good=purple
    color.bad=yellow
    allow.textmode=true
    how.nice.to.look=fairlyNice
kind: ConfigMap
metadata:
  creationTimestamp: 2016-02-18T18:34:05Z
  name: game-config
  namespace: default
  resourceVersion: "407"
  selfLink: /api/v1/namespaces/default/configmaps/game-config
  uid: 30944725-d66e-11e5-8cd0-68f728db1985
```

使用文件创建

刚才使用目录创建的时候我们 `—from-file` 指定的是一个目录，只要指定为一个文件就可以从单个文件中创建ConfigMap。

```
$ kubectl create configmap game-config-2 --from-file=docs/user-guide/configmap/kubectl/game.properties

$ kubectl get configmaps game-config-2 -o yaml
```

```yaml
apiVersion: v1
data:
  game-special-key: |
    enemies=aliens
    lives=3
    enemies.cheat=true
    enemies.cheat.level=noGoodRotten
    secret.code.passphrase=UUDDLRLRBABAS
    secret.code.allowed=true
    secret.code.lives=30
kind: ConfigMap
metadata:
  creationTimestamp: 2016-02-18T18:54:22Z
  name: game-config-3
  namespace: default
  resourceVersion: "530"
  selfLink: /api/v1/namespaces/default/configmaps/game-config-3
  uid: 05f8da22-d671-11e5-8cd0-68f728db1985
```

`–from-file` 这个参数可以使用多次，你可以使用两次分别指定上个实例中的那两个配置文件，效果就跟指定整个目录是一样的。

使用字面值创建

使用文字值创建，利用 `–from-literal` 参数传递配置信息，该参数可以使用多次，格式如下；

```
$ kubectl create configmap special-config --from-literal=special
.how=very --from-literal=special.type=charm

$ kubectl get configmaps special-config -o yaml
```

```yaml
apiVersion: v1
data:
  special.how: very
  special.type: charm
kind: ConfigMap
metadata:
  creationTimestamp: 2016-02-18T19:14:38Z
  name: special-config
  namespace: default
  resourceVersion: "651"
  selfLink: /api/v1/namespaces/default/configmaps/special-config
  uid: dadce046-d673-11e5-8cd0-68f728db1985
```

Pod中使用ConfigMap

使用**ConfigMap**来替代环境变量

ConfigMap可以被用来填入环境变量。看下下面的ConfigMap。

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: special-config
  namespace: default
data:
  special.how: very
  special.type: charm
```

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: env-config
  namespace: default
data:
  log_level: INFO
```

我们可以在Pod中这样使用ConfigMap：

```
apiVersion: v1
kind: Pod
metadata:
  name: dapi-test-pod
spec:
  containers:
    - name: test-container
      image: gcr.io/google_containers/busybox
      command: [ "/bin/sh", "-c", "env" ]
      env:
        - name: SPECIAL_LEVEL_KEY
          valueFrom:
            configMapKeyRef:
              name: special-config
              key: special.how
        - name: SPECIAL_TYPE_KEY
          valueFrom:
            configMapKeyRef:
              name: special-config
              key: special.type
      envFrom:
        - configMapRef:
            name: env-config
  restartPolicy: Never
```

这个Pod运行后会输出如下几行：

```
SPECIAL_LEVEL_KEY=very
SPECIAL_TYPE_KEY=charm
log_level=INFO
```

用**ConfigMap**设置命令行参数

ConfigMap也可以被使用来设置容器中的命令或者参数值。它使用的是Kubernetes的$(VAR_NAME)替换语法。我们看下下面这个ConfigMap。

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: special-config
  namespace: default
data:
  special.how: very
  special.type: charm
```

为了将ConfigMap中的值注入到命令行的参数里面，我们还要像前面那个例子一样使用环境变量替换语法 `${VAR_NAME)` 。（其实这个东西就是给Docker容器设置环境变量，以前我创建镜像的时候经常这么玩，通过docker run的时候指定-e参数修改镜像里的环境变量，然后docker的CMD命令再利用该$(VAR_NAME)通过sed来修改配置文件或者作为命令行启动参数。）

```
apiVersion: v1
kind: Pod
metadata:
  name: dapi-test-pod
spec:
  containers:
    - name: test-container
      image: gcr.io/google_containers/busybox
      command: [ "/bin/sh", "-c", "echo $(SPECIAL_LEVEL_KEY) $(S
PECIAL_TYPE_KEY)" ]
      env:
        - name: SPECIAL_LEVEL_KEY
          valueFrom:
            configMapKeyRef:
              name: special-config
              key: special.how
        - name: SPECIAL_TYPE_KEY
          valueFrom:
            configMapKeyRef:
              name: special-config
              key: special.type
  restartPolicy: Never
```

运行这个Pod后会输出：

```
very charm
```

通过数据卷插件使用ConfigMap

ConfigMap也可以在数据卷里面被使用。还是这个ConfigMap。

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: special-config
  namespace: default
data:
  special.how: very
  special.type: charm
```

在数据卷里面使用这个ConfigMap，有不同的选项。最基本的就是将文件填入数据卷，在这个文件中，键就是文件名，键值就是文件内容：

```
apiVersion: v1
kind: Pod
metadata:
  name: dapi-test-pod
spec:
  containers:
    - name: test-container
      image: gcr.io/google_containers/busybox
      command: [ "/bin/sh", "-c", "cat /etc/config/special.how" ]

      volumeMounts:
      - name: config-volume
        mountPath: /etc/config
  volumes:
    - name: config-volume
      configMap:
        name: special-config
  restartPolicy: Never
```

运行这个Pod的输出是 `very` 。

我们也可以在ConfigMap值被映射的数据卷里控制路径。

```
apiVersion: v1
kind: Pod
metadata:
  name: dapi-test-pod
spec:
  containers:
    - name: test-container
      image: gcr.io/google_containers/busybox
      command: [ "/bin/sh","-c","cat /etc/config/path/to/special
-key" ]
      volumeMounts:
      - name: config-volume
        mountPath: /etc/config
  volumes:
    - name: config-volume
      configMap:
        name: special-config
        items:
        - key: special.how
          path: path/to/special-key
  restartPolicy: Never
```

运行这个Pod后的结果是 `very` 。

ConfigMap的热更新

ConfigMap是用来存储配置文件的kubernetes资源对象，所有的配置内容都存储在etcd中，下文主要是探究 ConfigMap 的创建和更新流程，以及对 ConfigMap 更新后容器内挂载的内容是否同步更新的测试。

测试示例

假设我们在 `default` namespace 下有一个名为 `nginx-config` 的 ConfigMap，可以使用 `kubectl` 命令来获取：

```
$ kubectl get configmap nginx-config
NAME           DATA      AGE
nginx-config   1         99d
```

获取该ConfigMap的内容。

```
kubectl get configmap nginx-config -o yaml
```

```
apiVersion: v1
data:
  nginx.conf: |-
    worker_processes 1;

    events { worker_connections 1024; }

    http {
        sendfile on;

        server {
            listen 80;

            # a test endpoint that returns http 200s
            location / {
                proxy_pass http://httpstat.us/200;
                proxy_set_header  X-Real-IP  $remote_addr;
            }
        }

        server {
```

```
            listen 80;
            server_name api.hello.world;

            location / {
                proxy_pass http://l5d.default.svc.cluster.local;
                proxy_set_header Host $host;
                proxy_set_header Connection "";
                proxy_http_version 1.1;

                more_clear_input_headers 'l5d-ctx-*' 'l5d-dtab'
'l5d-sample';
            }
        }

        server {

            listen 80;
            server_name www.hello.world;

            location / {


                # allow 'employees' to perform dtab overrides
                if ($cookie_special_employee_cookie != "letmein")
 {
                  more_clear_input_headers 'l5d-ctx-*' 'l5d-dtab'
 'l5d-sample';
                }

                # add a dtab override to get people to our beta,
 world-v2
                set $xheader "";

                if ($cookie_special_employee_cookie ~* "dogfood")
 {
                  set $xheader "/host/world => /srv/world-v2;";
                }

                proxy_set_header 'l5d-dtab' $xheader;


                proxy_pass http://l5d.default.svc.cluster.local;
                proxy_set_header Host $host;
                proxy_set_header Connection "";
                proxy_http_version 1.1;
            }
        }
    }
kind: ConfigMap
metadata:
  creationTimestamp: 2017-08-01T06:53:17Z
  name: nginx-config
  namespace: default
```

```
  resourceVersion: "14925806"
  selfLink: /api/v1/namespaces/default/configmaps/nginx-config
  uid: 18d70527-7686-11e7-bfbd-8af1e3a7c5bd
```

ConfigMap中的内容是存储到etcd中的，然后查询etcd：

```
ETCDCTL_API=3 etcdctl get /registry/configmaps/default/nginx-con
fig -w json|python -m json.tool
```

注意使用 v3 版本的 etcdctl API，下面是输出结果：


```
{
    "count": 1,
    "header": {
        "cluster_id": 12091028579527406772,
        "member_id": 16557816780141026208,
        "raft_term": 36,
        "revision": 29258723
    },
    "kvs": [
        {
            "create_revision": 14925806,
            "key": "L3JlZ2lzdHJ5L2NvbmZpZ21hcHMvZGVmYXVsdC9uZ2lu
eC1jb25maWc=",
            "mod_revision": 14925806,
            "value": "azhzAAoPCgJ2MRIJQ29uZmlnTWFwEqQMClQKDG5naW
54LWNvbmZpZxIAGgdkZWZhdWx0IgAqJDE4ZDcwNTI3LTc2ODYtMTFlNy1iZmJkLT
hhZjFlM2E3YzViZDIAOABCCwjdyoDMBRC5ss54egASywsKCm5naW54LmNvbmYSvA
t3b3JrZXJfcHJvY2Vzc2VzIDE7CgpldmVudHMgeyB3b3JrZXJfY29ubmVjdGlvbn
MgMTAyNDsgfQoKaHR0cCB7CiAgICBzZW5kZmlsZSBvbjsKCiAgICBzZXJ2ZXIgew
ogICAgICAgIGxpc3RlbiA4MDsKCiAgICAgICAgIyBhIHRlc3QgZW5kcG9pbnQgdG
hhdCByZXR1cm5zIGh0dHAgMjAwcwogICAgICAgIGxvY2F0aW9uIC8gewogICAgIC
AgICAgICBwcm94eV9wYXNzIGh0dHA6Ly9odHRwc3RhdC51cy8yMDA7CiAgICAgIC
AgICAgIHByb3h5X3NldF9oZWFkZXIgIFgtUmVhbC1JUCAgJHJlbW90ZV9hZGRyOw
ogICAgICAgIH0KICAgIH0KCiAgICBzZXJ2ZXIgewoKICAgICAgICBsaXN0ZW4gOD
A7CiAgICAgICAgc2VydmVyX25hbWUgYXBpLmhlbGxvLndvcmxkOwoKICAgICAgIC
Bsb2NhdGlvbiAvIHsKICAgICAgICAgICAgcHJveHlfcGFzcyBodHRwOi8vbDVkLm
RlZmF1bHQuc3ZjLmNsdXN0ZXIubG9jYWw7CiAgICAgICAgICAgIHByb3h5X3NldF
9oZWFkZXIgSG9zdCAkaG9zdDsKICAgICAgICAgICAgcHJveHlfc2V0X2hlYWRlci
BDb25uZWN0aW9uICIiOwogICAgICAgICAgICBwcm94eV9odHRwX3ZlcnNpb24gMS
4xOwoKICAgICAgICAgICAgbW9yZV9jbGVhcl9pbnB1dF9oZWFkZXJzICdsNWQtY3
R4LSonICdsNWQtZHRhYicgJ2w1ZC1zYW1wbGUnOwogICAgICAgIH0KICAgIH0KCi
AgICBzZXJ2ZXIgewoKICAgICAgICBsaXN0ZW4gODA7CiAgICAgICAgc2VydmVyX2
5hbWUgd3d3LmhlbGxvLndvcmxkOwoKICAgICAgICBsb2NhdGlvbiAvIHsKCgogIC
AgICAgICAgICAjIGFsbG93ICdlbXBsb3llZXMnIHRvIHBlcmZvcm0gZHRhYiBvdm
VycmlkZXMKICAgICAgICAgICAgaWYgKCRjb29raWVfc3BlY2lhbF9lbXBsb3llZV
9jb29raWUgIT0gImxldG1laW4iKSB7CiAgICAgICAgICAgICAgbW9yZV9jbGVhcl
9pbnB1dF9oZWFkZXJzICdsNWQtY3R4LSonICdsNWQtZHRhYicgJ2w1ZC1zYW1wbG
```

```
UnOwogICAgICAgICAgICB9CgogICAgICAgICAgICAjIGFkZCBhIGR0YWIgb3Zlcn
JpZGUgdG8gZ2V0IHBlb3BsZSB0byBvdXIgYmV0YSwgd29ybGQtdjIKICAgICAgIC
AgICAgc2V0ICR4aGVhZGVyICIiOwoKICAgICAgICAgICAgaWYgKCRjb29raWVfc3
BlY2lhbF9lbXBsb3llZV9jb29raWUgfiogImRvZ2Zvb2QiKSB7CiAgICAgICAgIC
AgICAgc2V0ICR4aGVhZGVyICIvaG9zdC93b3JsZCA9PiAvc3J2L3dvcmxkLXYyOy
I7CiAgICAgICAgICAgIH0KCiAgICAgICAgICAgIHByb3h5X3NldF9oZWFkZXIgJ2
w1ZC1kdGFiJyAkeGhlYWRlcjsKCgogICAgICAgICAgICBwcm94eV9wYXNzIGh0dH
A6Ly9sNWQuZGVmYXVsdC5zdmMuY2x1c3Rlci5sb2NhbDsKICAgICAgICAgICAgcH
JveHlfc2V0X2hlYWRlciBIb3N0ICRob3N0OwogICAgICAgICAgICBwcm94eV9zZX
RfaGVhZGVyIENvbm5lY3Rpb24gIiI7CiAgICAgICAgICAgIHByb3h5X2h0dHBfdm
Vyc2lvbiAxLjE7CiAgICAgICAgfQogICAgfQp9GgAiAA==",
                "version": 1
            }
        ]
    }
```


其中的value就是 `nginx.conf` 配置文件的内容。

可以使用base64解码查看具体值，关于etcdctl的使用请参考使用etcdctl访问kuberentes数据。

代码

ConfigMap 结构体的定义：

```go
// ConfigMap holds configuration data for pods to consume.
type ConfigMap struct {
    metav1.TypeMeta `json:",inline"`
    // Standard object's metadata.
    // More info: http://releases.k8s.io/HEAD/docs/devel/api-conventions.md#metadata
    // +optional
    metav1.ObjectMeta `json:"metadata,omitempty" protobuf:"bytes,1,opt,name=metadata"`

    // Data contains the configuration data.
    // Each key must be a valid DNS_SUBDOMAIN with an optional leading dot.
    // +optional
    Data map[string]string `json:"data,omitempty" protobuf:"bytes,2,rep,name=data"`
}
```

在 `staging/src/k8s.io/client-go/kubernetes/typed/core/v1/configmap.go` 中ConfigMap 的接口定义：

```
// ConfigMapInterface has methods to work with ConfigMap resourc
es.
type ConfigMapInterface interface {
    Create(*v1.ConfigMap) (*v1.ConfigMap, error)
    Update(*v1.ConfigMap) (*v1.ConfigMap, error)
    Delete(name string, options *meta_v1.DeleteOptions) error
    DeleteCollection(options *meta_v1.DeleteOptions, listOptions
 meta_v1.ListOptions) error
    Get(name string, options meta_v1.GetOptions) (*v1.ConfigMap,
error)
    List(opts meta_v1.ListOptions) (*v1.ConfigMapList, error)
    Watch(opts meta_v1.ListOptions) (watch.Interface, error)
    Patch(name string, pt types.PatchType, data []byte, subresou
rces ...string) (result *v1.ConfigMap, err error)
    ConfigMapExpansion
}
```

在 `staging/src/k8s.io/client-go/kubernetes/typed/core/v1/configmap.go` 中创建 ConfigMap 的方法如下:

```
// Create takes the representation of a configMap and creates it
.  Returns the server's representation of the configMap, and an
error, if there is any.
func (c *configMaps) Create(configMap *v1.ConfigMap) (result *v1.
ConfigMap, err error) {
    result = &v1.ConfigMap{}
    err = c.client.Post().
        Namespace(c.ns).
        Resource("configmaps").
        Body(configMap).
        Do().
        Into(result)
    return
}
```

通过 RESTful 请求在 etcd 中存储 ConfigMap 的配置，该方法中设置了资源对象的 namespace 和 HTTP 请求中的 body，执行后将请求结果保存到 result 中返回给调用者。

注意 **Body** 的结构

```
// Body makes the request use obj as the body. Optional.
// If obj is a string, try to read a file of that name.
// If obj is a []byte, send it directly.
// If obj is an io.Reader, use it directly.
// If obj is a runtime.Object, marshal it correctly, and set Con
tent-Type header.
// If obj is a runtime.Object and nil, do nothing.
// Otherwise, set an error.
```

创建 ConfigMap RESTful 请求中的的 Body 中包含 `ObjectMeta` 和 `namespace`。

HTTP 请求中的结构体：

```
// Request allows for building up a request to a server in a cha
ined fashion.
// Any errors are stored until the end of your call, so you only
 have to
// check once.
type Request struct {
    // required
    client HTTPClient
    verb   string

    baseURL     *url.URL
    content     ContentConfig
    serializers Serializers

    // generic components accessible via method setters
    pathPrefix string
    subpath    string
    params     url.Values
    headers    http.Header

    // structural elements of the request that are part of the K
ubernetes API conventions
    namespace    string
    namespaceSet bool
    resource     string
    resourceName string
    subresource  string
    timeout      time.Duration

    // output
    err  error
    body io.Reader

    // This is only used for per-request timeouts, deadlines, an
d cancellations.
    ctx context.Context

    backoffMgr BackoffManager
    throttle   flowcontrol.RateLimiter
}
```

测试

分别测试使用 ConfigMap 挂载 Env 和 Volume 的情况。

更新使用ConfigMap挂载的Env

使用下面的配置创建 nginx 容器测试更新 ConfigMap 后容器内的环境变量是否也跟着更新。

```yaml
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: my-nginx
spec:
  replicas: 1
  template:
    metadata:
      labels:
        run: my-nginx
    spec:
      containers:
      - name: my-nginx
        image: harbor-001.jimmysong.io/library/nginx:1.9
        ports:
        - containerPort: 80
        envFrom:
        - configMapRef:
            name: env-config
---
apiVersion: v1
kind: ConfigMap
metadata:
  name: env-config
  namespace: default
data:
  log_level: INFO
```

获取环境变量的值

```
$ kubectl exec `kubectl get pods -l run=my-nginx  -o=name|cut -d "/" -f2` env|grep log_level
log_level=INFO
```

修改 ConfigMap

```
$ kubectl edit configmap env-config
```

修改 `log_level` 的值为 `DEBUG` 。

再次查看环境变量的值。

```
$ kubectl exec `kubectl get pods -l run=my-nginx  -o=name|cut -d
"/" -f2` env|grep log_level
log_level=INFO
```

实践证明修改 ConfigMap 无法更新容器中已注入的环境变量信息。

更新使用ConfigMap挂载的Volume

使用下面的配置创建 nginx 容器测试更新 ConfigMap 后容器内挂载的文件是否也跟着更新。

```yaml
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: my-nginx
spec:
  replicas: 1
  template:
    metadata:
      labels:
        run: my-nginx
    spec:
      containers:
      - name: my-nginx
        image: harbor-001.jimmysong.io/library/nginx:1.9
        ports:
        - containerPort: 80
      volumeMounts:
      - name: config-volume
        mountPath: /etc/config
      volumes:
        - name: config-volume
          configMap:
            name: special-config
---
apiVersion: v1
kind: ConfigMap
metadata:
  name: special-config
  namespace: default
data:
  log_level: INFO
```

```
$ kubectl exec `kubectl get pods -l run=my-nginx  -o=name|cut -d
"/" -f2` cat /tmp/log_level
INFO
```

修改 ConfigMap

```
$ kubectl edit configmap special-config
```

修改 `log_level` 的值为 `DEBUG` 。

等待大概10秒钟时间，再次查看环境变量的值。

```
$ kubectl exec `kubectl get pods -l run=my-nginx  -o=name|cut -d
"/" -f2` cat /tmp/log_level
DEBUG
```

我们可以看到使用 ConfigMap 方式挂载的 Volume 的文件中的内容已经变成了 `DEBUG` 。

ConfigMap 更新后滚动更新 Pod

更新 ConfigMap 目前并不会触发相关 Pod 的滚动更新，可以通过修改 pod annotations 的方式强制触发滚动更新。

```
$ kubectl patch deployment my-nginx --patch '{"spec": {"template
": {"metadata": {"annotations": {"version/config": "20180411" }}
}}}'
```

这个例子里我们在 `.spec.template.metadata.annotations` 中添加 `version/config` ，每次通过修改 `version/config` 来触发滚动更新。

总结

更新 ConfigMap 后：

- 使用该 ConfigMap 挂载的 Env 不会同步更新

- 使用该 ConfigMap 挂载的 Volume 中的数据需要一段时间（实测大概10秒）才能同步更新

ENV 是在容器启动的时候注入的，启动之后 kubernetes 就不会再改变环境变量的值，且同一个 namespace 中的 pod 的环境变量是不断累加的，参考 Kubernetes中的服务发现与docker容器间的环境变量传递源码探究。为了更新容器中使用 ConfigMap 挂载的配置，需要通过滚动更新 pod 的方式来强制重新挂载 ConfigMap。

参考

- Kubernetes 1.7 security in practice
- ConfigMap | kubernetes handbook - jimmysong.io
- 创建高可用ectd集群 | Kubernetes handbook - jimmysong.io
- Kubernetes中的服务发现与docker容器间的环境变量传递源码探究
- Automatically Roll Deployments When ConfigMaps or Secrets change

Volume

容器磁盘上的文件的生命周期是短暂的，这就使得在容器中运行重要应用时会出现一些问题。首先，当容器崩溃时，kubelet 会重启它，但是容器中的文件将丢失——容器以干净的状态（镜像最初的状态）重新启动。其次，在 `Pod` 中同时运行多个容器时，这些容器之间通常需要共享文件。Kubernetes 中的 `Volume` 抽象就很好的解决了这些问题。

建议先熟悉 pod。

背景

Docker 中也有一个 volume 的概念，尽管它稍微宽松一些，管理也很少。在 Docker 中，卷就像是磁盘或是另一个容器中的一个目录。它的生命周期不受管理，直到最近才有了 local-disk-backed 卷。Docker 现在提供了卷驱动程序，但是功能还非常有限（例如Docker1.7只允许每个容器使用一个卷驱动，并且无法给卷传递参数）。

另一方面，Kubernetes 中的卷有明确的寿命——与封装它的 Pod 相同。所以，卷的生命比 Pod 中的所有容器都长，当这个容器重启时数据仍然得以保存。当然，当 Pod 不再存在时，卷也将不复存在。也许更重要的是，Kubernetes 支持多种类型的卷，Pod 可以同时使用任意数量的卷。

卷的核心是目录，可能还包含了一些数据，可以通过 pod 中的容器来访问。该目录是如何形成的、支持该目录的介质以及其内容取决于所使用的特定卷类型。

要使用卷，需要为 pod 指定为卷（ `spec.volumes` 字段）以及将它挂载到容器的位置（ `spec.containers.volumeMounts` 字段）。

容器中的进程看到的是由其 Docker 镜像和卷组成的文件系统视图。 Docker 镜像位于文件系统层次结构的根目录，任何卷都被挂载在镜像的指定路径中。卷无法挂载到其他卷上或与其他卷有硬连接。Pod 中的每个容器都必须独立指定每个卷的挂载位置。

卷的类型

Kubernetes 支持以下类型的卷：

- `awsElasticBlockStore`
- `azureDisk`
- `azureFile`
- `cephfs`
- `csi`
- `downwardAPI`
- `emptyDir`
- `fc` (fibre channel)
- `flocker`
- `gcePersistentDisk`
- `gitRepo`
- `glusterfs`
- `hostPath`
- `iscsi`
- `local`
- `nfs`
- `persistentVolumeClaim`
- `projected`
- `portworxVolume`
- `quobyte`
- `rbd`
- `scaleIO`
- `secret`
- `storageos`
- `vsphereVolume`

我们欢迎额外贡献。

awsElasticBlockStore

`awsElasticBlockStore` 卷将Amazon Web Services（AWS）EBS Volume 挂载到您的容器中。与 `emptyDir` 类型会在删除 Pod 时被清除不同，EBS 卷的的内容会保留下来，仅仅是被卸载。这意味着 EBS 卷可以预先填充数据，并且可以在数据包之间“切换”数据。

重要提示：您必须使用 `aws ec2 create-volume` 或 AWS API 创建 EBS 卷，才能使用它。

使用 awsElasticBlockStore 卷时有一些限制：

- 运行 Pod 的节点必须是 AWS EC2 实例
- 这些实例需要与 EBS 卷位于相同的区域和可用区域
- EBS 仅支持卷和 EC2 实例的一对一的挂载

创建 EBS 卷

在 pod 中使用的 EBS 卷之前，您需要先创建它。

```
aws ec2 create-volume --availability-zone=eu-west-1a --size=10 --volume-type=gp2
```

确保区域与您启动集群的区域相匹配（并且检查大小和 EBS 卷类型是否适合您的使用！）

AWS EBS 示例配置

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: test-ebs
spec:
  containers:
  - image: k8s.gcr.io/test-webserver
    name: test-container
    volumeMounts:
    - mountPath: /test-ebs
      name: test-volume
  volumes:
  - name: test-volume
    # This AWS EBS volume must already exist.
    awsElasticBlockStore:
      volumeID: <volume-id>
      fsType: ext4
```

azureDisk

`AzureDisk` 用于将 Microsoft Azure Data Disk 挂载到 Pod 中。

更多细节可以在这里找到。

azureFile

`azureFile` 用于将 Microsoft Azure File Volume（SMB 2.1 和 3.0）挂载到 Pod 中。

更多细节可以在这里找到。

cephfs

`cephfs` 卷允许将现有的 CephFS 卷挂载到您的容器中。不像 `emptyDir`，当删除 Pod 时被删除，`cephfs` 卷的内容将被保留，卷仅仅是被卸载。这意味着 CephFS 卷可以预先填充数据，并且可以在数据包之间“切换”数据。 CephFS 可以被多个写设备同时挂载。

重要提示：您必须先拥有自己的 Ceph 服务器，然后才能使用它。

有关更多详细信息，请参见CephFS示例。

csi

CSI 代表容器存储接口，CSI 试图建立一个行业标准接口的规范，借助 CSI 容器编排系统（CO）可以将任意存储系统暴露给自己的容器工作负载。有关详细信息，请查看设计方案。

`csi` 卷类型是一种 in-tree 的 CSI 卷插件，用于 Pod 与在同一节点上运行的外部 CSI 卷驱动程序交互。部署 CSI 兼容卷驱动后，用户可以使用 `csi` 作为卷类型来挂载驱动提供的存储。

CSI 持久化卷支持是在 Kubernetes v1.9 中引入的，作为一个 alpha 特性，必须由集群管理员明确启用。换句话说，集群管理员需要在 apiserver、controller-manager 和 kubelet 组件的 “ `--feature-gates =` ” 标志中加上 “ `CSIPersistentVolume = true` ”。

CSI 持久化卷具有以下字段可供用户指定：

- `driver`：一个字符串值，指定要使用的卷驱动程序的名称。必须少于 63 个字符，并以一个字符开头。驱动程序名称可以包含 “ `.` ”、“ `-` ”、“ `_` ” 或数

字。

- `volumeHandle` ：一个字符串值，唯一标识从 CSI 卷插件的 `CreateVolume` 调用返回的卷名。随后在卷驱动程序的所有后续调用中使用卷句柄来引用该卷。
- `readOnly` ：一个可选的布尔值，指示卷是否被发布为只读。默认是 false。

downwardAPI

`downwardAPI` 卷用于使向下 API 数据（downward API data）对应用程序可用。它挂载一个目录，并将请求的数据写入纯文本文件。

参考 `downwardAPI` 卷示例查看详细信息。

emptyDir

当 Pod 被分配给节点时，首先创建 `emptyDir` 卷，并且只要该 Pod 在该节点上运行，该卷就会存在。正如卷的名字所述，它最初是空的。Pod 中的容器可以读取和写入 `emptyDir` 卷中的相同文件，尽管该卷可以挂载到每个容器中的相同或不同路径上。当出于任何原因从节点中删除 Pod 时， `emptyDir` 中的数据将被永久删除。

注意：容器崩溃不会从节点中移除 pod，因此 `emptyDir` 卷中的数据在容器崩溃时是安全的。

`emptyDir` 的用法有：

- 暂存空间，例如用于基于磁盘的合并排序
- 用作长时间计算崩溃恢复时的检查点
- Web服务器容器提供数据时，保存内容管理器容器提取的文件

Pod 示例

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: test-pd
spec:
  containers:
  - image: k8s.gcr.io/test-webserver
    name: test-container
    volumeMounts:
    - mountPath: /cache
      name: cache-volume
  volumes:
  - name: cache-volume
    emptyDir: {}
```

fc (fibre channel)

fc 卷允许将现有的 `fc` 卷挂载到 pod 中。您可以使用卷配置中的 `targetWWN` 参数指定单个或多个目标全球通用名称（World Wide Name）。如果指定了多个 WWN，则 targetWWN 期望这些 WWN 来自多路径连接。

重要提示：您必须配置 FC SAN 区域划分，并预先将这些 LUN（卷）分配并屏蔽到目标 WWN，以便 Kubernetes 主机可以访问它们。

参考 FC 示例获取详细信息。

flocker

Flocker 是一款开源的集群容器数据卷管理器。它提供了由各种存储后端支持的数据卷的管理和编排。

`flocker` 允许将 Flocker 数据集挂载到 pod 中。如果数据集在 Flocker 中不存在，则需要先使用 Flocker CLI 或使用 Flocker API 创建数据集。如果数据集已经存在，它将被 Flocker 重新连接到 pod 被调度的节点上。这意味着数据可以根据需要在数据包之间“切换”。

重要提示：您必须先运行自己的 Flocker 安装程序才能使用它。

参考 Flocker 示例获取更多详细信息。

gcePersistentDisk

`gcePersistentDisk` 卷将 Google Compute Engine（GCE）Persistent Disk 挂载到您的容器中。与删除 Pod 时删除的 `emptyDir` 不同，PD 的内容被保留，只是卸载了卷。这意味着 PD 可以预先填充数据，并且数据可以在 Pod 之间“切换”。

重要提示：您必须先使用 gcloud 或 GCE API 或 UI 创建一个 PD，然后才能使用它。

使用 `gcePersistentDisk` 时有一些限制：

- 运行 Pod 的节点必须是 GCE 虚拟机
- 那些虚拟机需要在与 PD 一样在 GCE 项目和区域中

PD 的一个特点是它们可以同时被多个用户以只读方式挂载。这意味着您可以预先使用您的数据集填充 PD，然后根据需要给多个 Pod 中并行提供。不幸的是，只能由单个消费者以读写模式挂载 PD，而不允许同时写入。 在由 ReplicationController 控制的 pod 上使用 PD 将会失败，除非 PD 是只读的或者副本数是 0 或 1。

创建 PD

在您在 pod 中使用 GCE PD 之前，需要先创建它。

```
gcloud compute disks create --size=500GB --zone=us-central1-a my
-data-disk
```

Pod 示例

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: test-pd
spec:
  containers:
  - image: k8s.gcr.io/test-webserver
    name: test-container
    volumeMounts:
    - mountPath: /test-pd
      name: test-volume
  volumes:
  - name: test-volume
    # This GCE PD must already exist.
    gcePersistentDisk:
      pdName: my-data-disk
      fsType: ext4
```

gitRepo

`gitRepo` 卷是一个可以演示卷插件功能的示例。它会挂载一个空目录并将 git 存储库克隆到您的容器中。将来，这样的卷可能会转移到一个更加分离的模型，而不是为每个这样的用例扩展 Kubernetes API。

下面是 gitRepo 卷示例：

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: server
spec:
  containers:
  - image: nginx
    name: nginx
    volumeMounts:
    - mountPath: /mypath
      name: git-volume
  volumes:
  - name: git-volume
    gitRepo:
      repository: "git@somewhere:me/my-git-repository.git"
      revision: "22f1d8406d464b0c0874075539c1f2e96c253775"
```

glusterfs

`glusterfs` 卷允许将 Glusterfs（一个开放源代码的网络文件系统）卷挂载到您的集群中。与删除 Pod 时删除的 `emptyDir` 不同， `glusterfs` 卷的内容将被保留，而卷仅仅被卸载。这意味着 glusterfs 卷可以预先填充数据，并且可以在数据包之间“切换”数据。 GlusterFS 可以同时由多个写入挂载。

重要提示：您必须先自行安装 GlusterFS，才能使用它。

有关更多详细信息，请参阅 GlusterFS 示例。

hostPath

`hostPath` 卷将主机节点的文件系统中的文件或目录挂载到集群中。该功能大多数 Pod 都用不到，但它为某些应用程序提供了一个强大的解决方法。

例如， `hostPath` 的用途如下：

- 运行需要访问 Docker 内部的容器；使用 `/var/lib/docker` 的 `hostPath`
- 在容器中运行 cAdvisor；使用 `/dev/cgroups` 的 `hostPath`
- 允许 pod 指定给定的 hostPath 是否应该在 pod 运行之前存在，是否应该创建，以及它应该以什么形式存在

除了所需的 `path` 属性之外，用户还可以为 `hostPath` 卷指定 `type` 。

`type` 字段支持以下值：

值	行为
	空字符串（默认）用于向后兼容，这意味着在挂载 hostPath 卷之前不会执行任何检查。
`DirectoryOrCreate`	如果在给定的路径上没有任何东西存在，那么将根据需要在那里创建一个空目录，权限设置为 0755，与 Kubelet 具有相同的组和所有权。
`Directory`	给定的路径下必须存在目录
`FileOrCreate`	如果在给定的路径上没有任何东西存在，那么会根据需要创建一个空文件，权限设置为 0644，与 Kubelet 具有相同的组和所有权。
`File`	给定的路径下必须存在文件
`Socket`	给定的路径下必须存在 UNIX 套接字
`CharDevice`	给定的路径下必须存在字符设备
`BlockDevice`	给定的路径下必须存在块设备

使用这种卷类型是请注意，因为：

- 由于每个节点上的文件都不同，具有相同配置（例如从 podTemplate 创建的）的 pod 在不同节点上的行为可能会有所不同
- 当 Kubernetes 按照计划添加资源感知调度时，将无法考虑 `hostPath` 使用的资源
- 在底层主机上创建的文件或目录只能由 root 写入。您需要在特权容器中以 root 身份运行进程，或修改主机上的文件权限以便写入 `hostPath` 卷

Pod 示例

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: test-pd
spec:
  containers:
  - image: k8s.gcr.io/test-webserver
    name: test-container
    volumeMounts:
    - mountPath: /test-pd
      name: test-volume
  volumes:
  - name: test-volume
    hostPath:
      # directory location on host
      path: /data
      # this field is optional
      type: Directory
```

iscsi

`iscsi` 卷允许将现有的 iSCSI（SCSI over IP）卷挂载到容器中。不像 `emptyDir` ，删除 Pod 时 `iscsi` 卷的内容将被保留，卷仅仅是被卸载。这意味着 iscsi 卷可以预先填充数据，并且这些数据可以在 pod 之间"切换"。

重要提示：必须先创建自己的 iSCSI 服务器，然后才能使用它。

iSCSI 的一个特点是它可以同时被多个用户以只读方式安装。这意味着您可以预先使用您的数据集填充卷，然后根据需要向多个额 pod 同时提供。不幸的是，iSCSI 卷只能由单个使用者以读写模式挂载——不允许同时写入。

有关更多详细信息，请参见 iSCSI示例。

local

这个 alpha 功能要求启用 `PersistentLocalVolumes` feature gate。

注意：从 1.9 开始， `VolumeScheduling` feature gate 也必须启用。

`local` 卷表示挂载的本地存储设备，如磁盘、分区或目录。

本地卷只能用作静态创建的 PersistentVolume。

与 HostPath 卷相比，local 卷可以以持久的方式使用，而无需手动将 pod 调度到节点上，因为系统会通过查看 PersistentVolume 上的节点关联性来了解卷的节点约束。

但是，local 卷仍然受底层节点的可用性影响，并不适用于所有应用程序。

以下是使用 `local` 卷的示例 PersistentVolume 规范：

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: example-pv
  annotations:
        "volume.alpha.kubernetes.io/node-affinity": '{
            "requiredDuringSchedulingIgnoredDuringExecution": {
                "nodeSelectorTerms": [
                    { "matchExpressions": [
                        { "key": "kubernetes.io/hostname",
                          "operator": "In",
                          "values": ["example-node"]
                        }
                    ]}
                 ]}
              }'
spec:
    capacity:
      storage: 100Gi
    accessModes:
    - ReadWriteOnce
    persistentVolumeReclaimPolicy: Delete
    storageClassName: local-storage
    local:
      path: /mnt/disks/ssd1
```

注意：本地 PersistentVolume 清理和删除需要手动干预，无外部提供程序。

从 1.9 开始，本地卷绑定可以被延迟，直到通过具有 StorageClass 中的 `WaitForFirstConsumer` 设置为 `volumeBindingMode` 的 pod 开始调度。请参阅示例。延迟卷绑定可确保卷绑定决策也可以使用任何其他节点约束（例如节点资源需求，节点选择器，pod 亲和性和 pod 反亲和性）进行评估。

有关 `local` 卷类型的详细信息，请参见本地持久化存储用户指南。

nfs

`nfs` 卷允许将现有的 NFS（网络文件系统）共享挂载到您的容器中。不像 `emptyDir` ，当删除 Pod 时， `nfs` 卷的内容被保留，卷仅仅是被卸载。这意味着 NFS 卷可以预填充数据，并且可以在 pod 之间“切换”数据。 NFS 可以被多个写入者同时挂载。

重要提示：您必须先拥有自己的 NFS 服务器才能使用它，然后才能使用它。

有关更多详细信息，请参见 NFS 示例。

persistentVolumeClaim

`persistentVolumeClaim` 卷用于将 PersistentVolume 挂载到容器中。PersistentVolumes 是在用户不知道特定云环境的细节的情况下“声明”持久化存储（例如 GCE PersistentDisk 或 iSCSI 卷）的一种方式。

有关更多详细信息，请参阅 PersistentVolumes 示例。

projected

`projected` 卷将几个现有的卷源映射到同一个目录中。

目前，可以映射以下类型的卷来源：

- `secret`
- `downwardAPI`
- `configMap`

所有来源都必须在与 pod 相同的命名空间中。有关更多详细信息，请参阅 all-in-one 卷设计文档。

带有 secret、downward API 和 configmap 的 pod

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: volume-test
spec:
  containers:
  - name: container-test
    image: busybox
    volumeMounts:
    - name: all-in-one
      mountPath: "/projected-volume"
      readOnly: true
  volumes:
  - name: all-in-one
    projected:
      sources:
      - secret:
          name: mysecret
          items:
            - key: username
              path: my-group/my-username
      - downwardAPI:
          items:
            - path: "labels"
              fieldRef:
                fieldPath: metadata.labels
            - path: "cpu_limit"
              resourceFieldRef:
                containerName: container-test
                resource: limits.cpu
      - configMap:
          name: myconfigmap
          items:
            - key: config
              path: my-group/my-config
```

使用非默认权限模式设置多个 secret 的示例 pod

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: volume-test
spec:
  containers:
  - name: container-test
    image: busybox
    volumeMounts:
    - name: all-in-one
      mountPath: "/projected-volume"
      readOnly: true
  volumes:
  - name: all-in-one
    projected:
      sources:
      - secret:
          name: mysecret
          items:
            - key: username
              path: my-group/my-username
      - secret:
          name: mysecret2
          items:
            - key: password
              path: my-group/my-password
              mode: 511
```

每个映射的卷来源在 `sources` 下的规格中列出。除了以下两个例外，参数几乎相同：

- 对于 secret，`secretName` 字段已经被更改为 `name` 以与 ConfigMap 命名一致。
- `defaultMode` 只能在映射级别指定，而不能针对每个卷源指定。但是，如上所述，您可以明确设置每个映射的 `mode` 。

portworxVolume

`portworxVolume` 是一个与 Kubernetes 一起，以超融合模式运行的弹性块存储层。Portwork 指纹存储在服务器中，基于功能的分层，以及跨多个服务器聚合容量。 Portworx 在虚拟机或裸机 Linux 节点上运行。

`portworxVolume` 可以通过 Kubernetes 动态创建，也可以在 Kubernetes pod 中预先设置和引用。

以下是一个引用预先配置的 PortworxVolume 的示例 pod：

```
apiVersion: v1
kind: Pod
metadata:
  name: test-portworx-volume-pod
spec:
  containers:
  - image: k8s.gcr.io/test-webserver
    name: test-container
    volumeMounts:
    - mountPath: /mnt
      name: pxvol
  volumes:
  - name: pxvol
    # This Portworx volume must already exist.
    portworxVolume:
      volumeID: "pxvol"
      fsType: "<fs-type>"
```

重要提示：在 pod 中使用之前，请确保您有一个名为 `pxvol` 的现有 PortworxVolume。

更多的细节和例子可以在这里找到。

quobyte

`quobyte` 卷允许将现有的 Quobyte 卷挂载到容器中。

重要提示：您必须先创建自己的 Quobyte 安装程序，然后才能使用它。

有关更多详细信息，请参见 Quobyte示例。

rbd

`rbd` 卷允许将 Rados Block Device 卷挂载到容器中。不像 `emptyDir`，删除 Pod 时 `rbd` 卷的内容被保留，卷仅仅被卸载。这意味着 RBD 卷可以预先填充数据，并且可以在 pod 之间“切换”数据。

重要提示：您必须先自行安装 Ceph，然后才能使用 RBD。

RBD 的一个特点是它可以同时为多个用户以只读方式挂载。这意味着可以预先使用您的数据集填充卷，然后根据需要同时为多个 pod 并行提供。不幸的是，RBD 卷只能由单个用户以读写模式安装——不允许同时写入。

有关更多详细信息，请参阅 RBD示例。

scaleIO

ScaleIO 是一个基于软件的存储平台，可以使用现有的硬件来创建可扩展的共享块网络存储集群。 `scaleIO` 卷插件允许已部署的 pod 访问现有的 ScaleIO 卷（或者它可以为持久性卷声明动态调配新卷，请参阅 ScaleIO 持久卷）。

重要提示：您必须有一个已经配置好的 ScaleIO 集群，并和创建的卷一同运行，然后才能使用它们。

以下是使用 ScaleIO 的示例 pod 配置：

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: pod-0
spec:
  containers:
  - image: k8s.gcr.io/test-webserver
    name: pod-0
    volumeMounts:
    - mountPath: /test-pd
      name: vol-0
  volumes:
  - name: vol-0
    scaleIO:
      gateway: https://localhost:443/api
      system: scaleio
      protectionDomain: sd0
      storagePool: sp1
      volumeName: vol-0
      secretRef:
        name: sio-secret
      fsType: xfs
```

有关更多详细信息，请参阅 ScaleIO 示例。

secret

`secret` 卷用于将敏感信息（如密码）传递到 pod。您可以将 secret 存储在 Kubernetes API 中，并将它们挂载为文件，以供 Pod 使用，而无需直接连接到 Kubernetes。 `secret` 卷由 tmpfs（一个 RAM 支持的文件系统）支持，所以它们永远不会写入非易失性存储器。

重要提示：您必须先在 Kubernetes API 中创建一个 secret，然后才能使用它。

Secret 在这里被更详细地描述。

storageOS

`storageos` 卷允许将现有的 StorageOS 卷挂载到容器中。

StorageOS 在 Kubernetes 环境中以容器方式运行，使本地或附加存储可以从 Kubernetes 集群中的任何节点访问。可以复制数据以防止节点故障。精简配置和压缩可以提高利用率并降低成本。

StorageOS 的核心是为容器提供块存储，可以通过文件系统访问。

StorageOS 容器需要 64 位 Linux，没有额外的依赖关系。可以使用免费的开发者许可证。

重要提示：您必须在每个要访问 StorageOS 卷的节点上运行 StorageOS 容器，或者为该池提供存储容量。相关的安装说明，请参阅 StorageOS文档。

```yaml
apiVersion: v1
kind: Pod
metadata:
  labels:
    name: redis
    role: master
  name: test-storageos-redis
spec:
  containers:
    - name: master
      image: kubernetes/redis:v1
      env:
        - name: MASTER
          value: "true"
      ports:
        - containerPort: 6379
      volumeMounts:
        - mountPath: /redis-master-data
          name: redis-data
  volumes:
    - name: redis-data
      storageos:
        # The `redis-vol01` volume must already exist within StorageOS in the `default` namespace.
        volumeName: redis-vol01
        fsType: ext4
```

有关更多信息，包括动态配置和持久化卷声明，请参阅 StorageOS 示例。

vsphereVolume

先决条件：配置了 vSphere Cloud Provider 的 Kubernetes。有关云提供商的配置，请参阅 vSphere 入门指南。

`vsphereVolume` 用于将 vSphere VMDK 卷挂载到 Pod 中。卷的内容在卸载时会被保留。支持 VMFS 和 VSAN 数据存储。

重要提示：在 Pod 中使用它之前，您必须使用以下一种方法创建 VMDK。

创建 VMDK 卷

选择以下方法之一来创建 VMDK。

首先进入 ESX，然后使用以下命令创建一个 VMDK：

```
vmkfstools -c 2G /vmfs/volumes/DatastoreName/volumes/myDisk.vmdk
```

使用下列命令创建一个 VMDK：

```
vmware-vdiskmanager -c -t 0 -s 40GB -a lsilogic myDisk.vmdk
```

vSphere VMDK 示例配置

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: test-vmdk
spec:
  containers:
  - image: k8s.gcr.io/test-webserver
    name: test-container
    volumeMounts:
    - mountPath: /test-vmdk
      name: test-volume
  volumes:
  - name: test-volume
    # This VMDK volume must already exist.
    vsphereVolume:
      volumePath: "[DatastoreName] volumes/myDisk"
      fsType: ext4
```

更多的例子可以在这里找到。

使用 subPath

有时，在单个容器中共享一个卷用于多个用途是有用的。`volumeMounts.subPath` 属性可用于在引用的卷内而不是其根目录中指定子路径。

下面是一个使用单个共享卷的 LAMP 堆栈（Linux Apache Mysql PHP）的示例。HTML 内容被映射到它的 html 目录，数据库将被存储在它的 mysql 目录中：

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-lamp-site
spec:
    containers:
    - name: mysql
      image: mysql
      env:
      - name: MYSQL_ROOT_PASSWORD
        value: "rootpasswd"
      volumeMounts:
      - mountPath: /var/lib/mysql
        name: site-data
        subPath: mysql
    - name: php
      image: php:7.0-apache
      volumeMounts:
      - mountPath: /var/www/html
        name: site-data
        subPath: html
    volumes:
    - name: site-data
      persistentVolumeClaim:
        claimName: my-lamp-site-data
```

资源

`emptyDir` 卷的存储介质（磁盘、SSD 等）由保存在 kubelet 根目录的文件系统的介质（通常是 `/var/lib/kubelet`）决定。 `emptyDir` 或 `hostPath` 卷可占用多少空间并没有限制，容器之间或 Pod 之间也没有隔离。

在将来，我们预计 `emptyDir` 和 `hostPath` 卷将能够使用 resource 规范请求一定的空间，并选择要使用的介质，适用于具有多种媒体类型的集群。

Out-of-Tree 卷插件

除了之前列出的卷类型之外，存储供应商可以创建自定义插件而不将其添加到 Kubernetes 存储库中。可以通过使用 `FlexVolume` 插件来实现。

`FlexVolume` 使用户能够将供应商卷挂载到容器中。供应商插件是使用驱动程序实现的，该驱动程序支持由 `FlexVolume` API定义的一系列卷命令。驱动程序必须安装在每个节点的预定义卷插件路径中。

更多细节可以在这里找到。

挂载传播

注意：挂载传播是 Kubernetes 1.8 中的一个 alpha 特性，在将来的版本中可能会重新设计甚至删除。

挂载传播允许将由容器挂载的卷共享到同一个 Pod 中的其他容器上，甚至是同一节点上的其他 Pod。

如果禁用 MountPropagation 功能，则不会传播 pod 中的卷挂载。也就是说，容器按照 Linux内核文档中所述的 `private` 挂载传播运行。

要启用此功能，请在 `--feature-gates` 命令行选项中指定 `MountPropagation = true` 。启用时，容器的 `volumeMounts` 字段有一个新的 `mountPropagation` 子字段。它的值为：

- `HostToContainer` ：此卷挂载将接收所有后续挂载到此卷或其任何子目录的挂载。这是 MountPropagation 功能启用时的默认模式。

  同样的，如果任何带有 `Bidirectional` 挂载传播的 pod 挂载到同一个卷上，带有 `HostToContainer` 挂载传播的容器将会看到它。

  该模式等同于Linux内核文档中描述的 `rslave` 挂载传播。

- `Bidirectional` 卷挂载与 `HostToContainer` 挂载相同。另外，由容器创建的所有卷挂载将被传播回主机和所有使用相同卷的容器的所有容器。

  此模式的一个典型用例是带有 Flex 卷驱动器或需要使用 HostPath 卷在主机上挂载某些内容的 pod。

  该模式等同于 Linux内核文档中所述的 `rshared` 挂载传播。

小心：双向挂载传播可能是危险的。它可能会损坏主机操作系统，因此只能在特权容器中使用。强烈建议熟悉 Linux 内核行为。另外，容器在 Pod 中创建的任何卷挂载必须在容器终止时销毁（卸载）。

参考

- https://kubernetes.io/docs/concepts/storage/volumes/

- 使用持久化卷来部署 WordPress 和 MySQL

Persistent Volume（持久化卷）

本文档介绍了 Kubernetes 中 `PersistentVolume` 的当前状态。建议您在阅读本文档前先熟悉 volume。

介绍

对于管理计算资源来说，管理存储资源明显是另一个问题。`PersistentVolume` 子系统为用户和管理员提供了一个 API，该 API 将如何提供存储的细节抽象了出来。为此，我们引入两个新的 API 资源：`PersistentVolume` 和 `PersistentVolumeClaim`。

`PersistentVolume`（PV）是由管理员设置的存储，它是群集的一部分。就像节点是集群中的资源一样，PV 也是集群中的资源。 PV 是 Volume 之类的卷插件，但具有独立于使用 PV 的 Pod 的生命周期。此 API 对象包含存储实现的细节，即 NFS、iSCSI 或特定于云供应商的存储系统。

`PersistentVolumeClaim`（PVC）是用户存储的请求。它与 Pod 相似。Pod 消耗节点资源，PVC 消耗 PV 资源。Pod 可以请求特定级别的资源（CPU 和内存）。声明可以请求特定的大小和访问模式（例如，可以以读/写一次或 只读多次模式挂载）。

虽然 `PersistentVolumeClaims` 允许用户使用抽象存储资源，但用户需要具有不同性质（例如性能）的 `PersistentVolume` 来解决不同的问题。集群管理员需要能够提供各种各样的 `PersistentVolume`，这些 `PersistentVolume` 的大小和访问模式可以各有不同，但不需要向用户公开实现这些卷的细节。对于这些需求，`StorageClass` 资源可以实现。

请参阅工作示例的详细过程。

卷和声明的生命周期

PV 属于集群中的资源。PVC 是对这些资源的请求，也作为对资源的请求的检查。PV 和 PVC 之间的相互作用遵循这样的生命周期：

配置（Provision）

有两种方式来配置 PV：静态或动态。

静态

集群管理员创建一些 PV。它们带有可供群集用户使用的实际存储的细节。它们存在于 Kubernetes API 中，可用于消费。

动态

当管理员创建的静态 PV 都不匹配用户的 `PersistentVolumeClaim` 时，集群可能会尝试动态地为 PVC 创建卷。此配置基于 `StorageClasses`：PVC 必须请求存储类，并且管理员必须创建并配置该类才能进行动态创建。声明该类为 `""` 可以有效地禁用其动态配置。

要启用基于存储级别的动态存储配置，集群管理员需要启用 API server 上的 `DefaultStorageClass` 准入控制器。例如，通过确保 `DefaultStorageClass` 位于 API server 组件的 `--admission-control` 标志，使用逗号分隔的有序值列表中，可以完成此操作。有关 API server 命令行标志的更多信息，请检查 kube-apiserver 文档。

绑定

再动态配置的情况下，用户创建或已经创建了具有特定存储量的 `PersistentVolumeClaim` 以及某些访问模式。master 中的控制环路监视新的 PVC，寻找匹配的 PV（如果可能），并将它们绑定在一起。如果为新的 PVC 动态调配 PV，则该环路将始终将该 PV 绑定到 PVC。否则，用户总会得到他们所请求的存储，但是容量可能超出要求的数量。一旦 PV 和 PVC 绑定后，`PersistentVolumeClaim` 绑定是排他性的，不管它们是如何绑定的。 PVC 跟 PV 绑定是一对一的映射。

如果没有匹配的卷，声明将无限期地保持未绑定状态。随着匹配卷的可用，声明将被绑定。例如，配置了许多 50Gi PV的集群将不会匹配请求 100Gi 的PVC。将 100Gi PV 添加到群集时，可以绑定 PVC。

使用

Pod 使用声明作为卷。集群检查声明以查找绑定的卷并为集群挂载该卷。对于支持多种访问模式的卷，用户指定在使用声明作为容器中的卷时所需的模式。

用户进行了声明，并且该声明是绑定的，则只要用户需要，绑定的 PV 就属于该用户。用户通过在 Pod 的 volume 配置中包含 `persistentVolumeClaim` 来调度 Pod 并访问用户声明的 PV。请参阅下面的语法细节。

持久化卷声明的保护

PVC 保护的目的是确保由 pod 正在使用的 PVC 不会从系统中移除，因为如果被移除的话可能会导致数据丢失。

注意：当 pod 状态为 `Pending` 并且 pod 已经分配给节点或 pod 为 `Running` 状态时，PVC 处于活动状态。

当启用PVC 保护 alpha 功能时，如果用户删除了一个 pod 正在使用的 PVC，则该 PVC 不会被立即删除。PVC 的删除将被推迟，直到 PVC 不再被任何 pod 使用。

您可以看到，当 PVC 的状态为 `Teminatiing` 时，PVC 受到保护，`Finalizers` 列表中包含 `kubernetes.io/pvc-protection`：

```
kubectl described pvc hostpath
Name:          hostpath
Namespace:     default
StorageClass:  example-hostpath
Status:        Terminating
Volume:
Labels:        <none>
Annotations:   volume.beta.kubernetes.io/storage-class=example-h
ostpath
               volume.beta.kubernetes.io/storage-provisioner=exa
mple.com/hostpath
Finalizers:    [kubernetes.io/pvc-protection]
...
```

回收

用户用完 volume 后，可以从允许回收资源的 API 中删除 PVC 对象。`PersistentVolume` 的回收策略告诉集群在存储卷声明释放后应如何处理该卷。目前，volume 的处理策略有保留、回收或删除。

保留

保留回收策略允许手动回收资源。当 `PersistentVolumeClaim` 被删除时， `PersistentVolume` 仍然存在，volume 被视为“已释放”。但是由于前一个声明人的数据仍然存在，所以还不能马上进行其他声明。管理员可以通过以下步骤手动回收卷。

1. 删除 `PersistentVolume` 。在删除 PV 后，外部基础架构中的关联存储资产（如 AWS EBS、GCE PD、Azure Disk 或 Cinder 卷）仍然存在。
2. 手动清理相关存储资产上的数据。
3. 手动删除关联的存储资产，或者如果要重新使用相同的存储资产，请使用存储资产定义创建新的 `PersistentVolume` 。

回收

如果存储卷插件支持，回收策略会在 volume上执行基本擦除（ `rm -rf / thevolume / *` ），可被再次声明使用。

但是，管理员可以使用如此处所述的 Kubernetes controller manager 命令行参数来配置自定义回收站 pod 模板。自定义回收站 pod 模板必须包含 `volumes` 规范，如下面的示例所示：

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: pv-recycler
  namespace: default
spec:
  restartPolicy: Never
  volumes:
  - name: vol
    hostPath:
      path: /any/path/it/will/be/replaced
  containers:
  - name: pv-recycler
    image: "k8s.gcr.io/busybox"
    command: ["/bin/sh", "-c", "test -e /scrub && rm -rf /scrub/..?* /scrub/.[!.]* /scrub/*  && test -z \"$(ls -A /scrub)\" || exit 1"]
    volumeMounts:
    - name: vol
      mountPath: /scrub
```

但是，`volumes` 部分的自定义回收站模块中指定的特定路径将被替换为正在回收的卷的特定路径。

删除

对于支持删除回收策略的卷插件，删除操作将从 Kubernetes 中删除 `PersistentVolume` 对象，并删除外部基础架构（如 AWS EBS、GCE PD、Azure Disk 或 Cinder 卷）中的关联存储资产。动态配置的卷继承其 `StorageClass` 的回收策略，默认为 Delete。管理员应该根据用户的期望来配置 `StorageClass`，否则就必须要在 PV 创建后进行编辑或修补。请参阅更改 PersistentVolume 的回收策略。

扩展持久化卷声明

Kubernetes 1.8 增加了对扩展持久化存储卷的 Alpha 支持。在 v1.9 中，以下持久化卷支持扩展持久化卷声明：

- gcePersistentDisk
- awsElasticBlockStore
- Cinder
- glusterfs
- rbd

管理员可以通过将 `ExpandPersistentVolumes` 特性门设置为true来允许扩展持久卷声明。管理员还应该启用 `PersistentVolumeClaimResize` 准入控制插件来执行对可调整大小的卷的其他验证。

一旦 `PersistentVolumeClaimResize` 准入插件已打开，将只允许其 `allowVolumeExpansion` 字段设置为 true 的存储类进行大小调整。

```yaml
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: gluster-vol-default
provisioner: kubernetes.io/glusterfs
parameters:
  resturl: "http://192.168.10.100:8080"
  restuser: ""
  secretNamespace: ""
  secretName: ""
allowVolumeExpansion: true
```

一旦功能门和前述准入插件打开后，用户就可以通过简单地编辑声明以请求更大的 `PersistentVolumeClaim` 卷。这反过来将触发 `PersistentVolume` 后端的卷扩展。

在任何情况下都不会创建新的 `PersistentVolume` 来满足声明。 Kubernetes 将尝试调整现有 volume 来满足声明的要求。

对于扩展包含文件系统的卷，只有在 ReadWrite 模式下使用 `PersistentVolumeClaim` 启动新的 Pod 时，才会执行文件系统调整大小。换句话说，如果正在扩展的卷在 pod 或部署中使用，则需要删除并重新创建要进行文件系统调整大小的pod。此外，文件系统调整大小仅适用于以下文件系统类型：

- XFS
- Ext3、Ext4

注意：扩展 EBS 卷是一个耗时的操作。另外，每6个小时有一个修改卷的配额。

持久化卷类型

`PersistentVolume` 类型以插件形式实现。Kubernetes 目前支持以下插件类型：

- GCEPersistentDisk
- AWSElasticBlockStore
- AzureFile
- AzureDisk
- FC (Fibre Channel)
- FlexVolume
- Flocker
- NFS
- iSCSI
- RBD (Ceph Block Device)
- CephFS
- Cinder (OpenStack block storage)
- Glusterfs
- VsphereVolume
- Quobyte Volumes
- HostPath （仅限于但节点测试—— 不会以任何方式支持本地存储，也无法在多节点集群中工作）

- VMware Photon
- Portworx Volumes
- ScaleIO Volumes
- StorageOS

原始块支持仅适用于以上这些插件。

持久化卷

每个 PV 配置中都包含一个 sepc 规格字段和一个 status 卷状态字段。

```yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv0003
spec:
  capacity:
    storage: 5Gi
  volumeMode: Filesystem
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Recycle
  storageClassName: slow
  mountOptions:
    - hard
    - nfsvers=4.1
  nfs:
    path: /tmp
    server: 172.17.0.2
```

容量

通常，PV 将具有特定的存储容量。这是使用 PV 的容量属性设置的。查看 Kubernetes 资源模型 以了解 `capacity` 预期。

目前，存储大小是可以设置或请求的唯一资源。未来的属性可能包括 IOPS、吞吐量等。

卷模式

在 v1.9 之前，所有卷插件的默认行为是在持久卷上创建一个文件系统。在 v1.9 中，用户可以指定一个 volumeMode，除了文件系统之外，它现在将支持原始块设备。 volumeMode 的有效值可以是“Filesystem”或“Block”。如果未指定，volumeMode 将默认为“Filesystem”。这是一个可选的 API 参数。

注意：该功能在 V1.9 中是 alpha的，未来可能会更改。

访问模式

`PersistentVolume` 可以以资源提供者支持的任何方式挂载到主机上。如下表所示，供应商具有不同的功能，每个 PV 的访问模式都将被设置为该卷支持的特定模式。例如，NFS 可以支持多个读/写客户端，但特定的 NFS PV 可能以只读方式导出到服务器上。每个 PV 都有一套自己的用来描述特定功能的访问模式。

存储模式包括：

- ReadWriteOnce——该卷可以被单个节点以读/写模式挂载
- ReadOnlyMany——该卷可以被多个节点以只读模式挂载
- ReadWriteMany——该卷可以被多个节点以读/写模式挂载

在命令行中，访问模式缩写为：

- RWO - ReadWriteOnce
- ROX - ReadOnlyMany
- RWX - ReadWriteMany

> 重要！一个卷一次只能使用一种访问模式挂载，即使它支持很多访问模式。例如，GCEPersistentDisk 可以由单个节点作为 ReadWriteOnce 模式挂载，或由多个节点以 ReadOnlyMany 模式挂载，但不能同时挂载。

Volume 插件	ReadWriteOnce	ReadOnlyMany	ReadWriteMa
AWSElasticBlockStore	✓	-	-
AzureFile	✓	✓	✓
AzureDisk	✓	-	-
CephFS	✓	✓	✓
Cinder	✓	-	-
FC	✓	✓	-
FlexVolume	✓	✓	-
Flocker	✓	-	-
GCEPersistentDisk	✓	✓	-
Glusterfs	✓	✓	✓
HostPath	✓	-	-
iSCSI	✓	✓	-
PhotonPersistentDisk	✓	-	-
Quobyte	✓	✓	✓
NFS	✓	✓	✓
RBD	✓	✓	-
VsphereVolume	✓	-	- （当 pod 并 时有效）
PortworxVolume	✓	-	✓
ScaleIO	✓	✓	-
StorageOS	✓	-	-

类

PV 可以具有一个类，通过将 `storageClassName` 属性设置为 StorageClass 的名称来指定该类。一个特定类别的 PV 只能绑定到请求该类别的 PVC。没有 `storageClassName` 的 PV 就没有类，它只能绑定到不需要特定类的 PVC。

过去，使用的是 `volume.beta.kubernetes.io/storage-class` 注解而不是 `storageClassName` 属性。这个注解仍然有效，但是将来的 Kubernetes 版本中将会完全弃用它。

回收策略

当前的回收策略包括：

- Retain（保留）——手动回收
- Recycle（回收）——基本擦除（ `rm -rf /thevolume/*` ）
- Delete（删除）——关联的存储资产（例如 AWS EBS、GCE PD、Azure Disk 和 OpenStack Cinder 卷）将被删除

当前，只有 NFS 和 HostPath 支持回收策略。AWS EBS、GCE PD、Azure Disk 和 Cinder 卷支持删除策略。

挂载选项

Kubernetes 管理员可以指定在节点上为挂载持久卷指定挂载选项。

注意：不是所有的持久化卷类型都支持挂载选项。

以下卷类型支持挂载选项：

- GCEPersistentDisk
- AWSElasticBlockStore
- AzureFile
- AzureDisk
- NFS
- iSCSI
- RBD （Ceph Block Device）
- CephFS
- Cinder （OpenStack 卷存储）
- Glusterfs
- VsphereVolume
- Quobyte Volumes
- VMware Photon

挂载选项没有校验，如果挂载选项无效则挂载失败。

过去，使用 `volume.beta.kubernetes.io/mount-options` 注解而不是 `mountOptions` 属性。这个注解仍然有效，但在将来的 Kubernetes 版本中它将会被完全弃用。

状态

卷可以处于以下的某种状态：

- Available（可用）——一块空闲资源还没有被任何声明绑定
- Bound（已绑定）——卷已经被声明绑定
- Released（已释放）——声明被删除，但是资源还未被集群重新声明
- Failed（失败）——该卷的自动回收失败

命令行会显示绑定到 PV 的 PVC 的名称。

PersistentVolumeClaim

每个 PVC 中都包含一个 spec 规格字段和一个 status 声明状态字段。

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: myclaim
spec:
  accessModes:
    - ReadWriteOnce
  volumeMode: Filesystem
  resources:
    requests:
      storage: 8Gi
  storageClassName: slow
  selector:
    matchLabels:
      release: "stable"
    matchExpressions:
      - {key: environment, operator: In, values: [dev]}
```

访问模式

在请求具有特定访问模式的存储时，声明使用与卷相同的约定。

卷模式

声明使用与卷相同的约定，指示将卷作为文件系统或块设备使用。

资源

像 pod 一样，声明可以请求特定数量的资源。在这种情况下，请求是用于存储的。相同的资源模型适用于卷和声明。

选择器

声明可以指定一个标签选择器来进一步过滤该组卷。只有标签与选择器匹配的卷可以绑定到声明。选择器由两个字段组成：

- matchLabels：volume 必须有具有该值的标签
- matchExpressions：这是一个要求列表，通过指定关键字，值列表以及与关键字和值相关的运算符组成。有效的运算符包括 In、NotIn、Exists 和 DoesNotExist。

所有来自 `matchLabels` 和 `matchExpressions` 的要求都被“与”在一起——它们必须全部满足才能匹配。

类

声明可以通过使用属性 `storageClassName` 指定 StorageClass 的名称来请求特定的类。只有所请求的类与 PVC 具有相同 `storageClassName` 的 PV 才能绑定到 PVC。

PVC 不一定要请求类。其 `storageClassName` 设置为 `""` 的 PVC 始终被解释为没有请求类的 PV，因此只能绑定到没有类的 PV（没有注解或 `""`）。没有 `storageClassName` 的 PVC 根据是否打开 `DefaultStorageClass` 准入控制插件，集群对其进行不同处理。

- 如果打开了准入控制插件，管理员可以指定一个默认的 `StorageClass`。所有没有 `StorageClassName` 的 PVC 将被绑定到该默认的 PV。通过在 `StorageClass` 对象中将注解 `storageclassclass.ubernetes.io/is-default-class` 设置为 “true” 来指定默认的 `StorageClass`。如果管理员没有指定缺省值，那么集群会响应 PVC 创建，就好像关闭了准入控制插件一样。如果指定了多个默认值，则准入控制插件将禁止所有 PVC 创建。
- 如果准入控制插件被关闭，则没有默认 `StorageClass` 的概念。所有没有 `storageClassName` 的 PVC 只能绑定到没有类的 PV。在这种情况下，没有 `storageClassName` 的 PVC 的处理方式与 `storageClassName` 设置为

`""` 的 PVC 的处理方式相同。

根据安装方法的不同，默认的 `StorageClass` 可以在安装过程中通过插件管理器部署到 Kubernetes 集群。

当 PVC 指定了 `selector`，除了请求一个 `StorageClass` 之外，这些需求被“与”在一起：只有被请求的类的 PV 具有和被请求的标签才可以被绑定到 PVC。

注意：目前，具有非空 `selector` 的 PVC 不能为其动态配置 PV。

过去，使用注解 `volume.beta.kubernetes.io/storage-class` 而不是 `storageClassName` 属性。这个注解仍然有效，但是在未来的 Kubernetes 版本中不会支持。

声明作为卷

通过将声明用作卷来访问存储。声明必须与使用声明的 pod 存在于相同的命名空间中。集群在 pod 的命名空间中查找声明，并使用它来获取支持声明的 `PersistentVolume`。该卷然后被挂载到主机的 pod 上。

```yaml
kind: Pod
apiVersion: v1
metadata:
  name: mypod
spec:
  containers:
    - name: myfrontend
      image: dockerfile/nginx
      volumeMounts:
      - mountPath: "/var/www/html"
        name: mypd
  volumes:
    - name: mypd
      persistentVolumeClaim:
        claimName: myclaim
```

命名空间注意点

`PersistentVolumes` 绑定是唯一的，并且由于 `PersistentVolumeClaims` 是命名空间对象，因此只能在一个命名空间内挂载具有“多个”模式（`ROX`、`RWX`）的声明。

原始块卷支持

原始块卷的静态配置在 v1.9 中作为 alpha 功能引入。由于这个改变，需要一些新的 API 字段来使用该功能。目前，Fibre Channl 是支持该功能的唯一插件。

使用原始块卷作为持久化卷

```yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: block-pv
spec:
  capacity:
    storage: 10Gi
  accessModes:
    - ReadWriteOnce
  volumeMode: Block
  persistentVolumeReclaimPolicy: Retain
  fc:
    targetWWNs: ["50060e801049cfd1"]
    lun: 0
    readOnly: false
```

持久化卷声明请求原始块卷

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: block-pvc
spec:
  accessModes:
    - ReadWriteOnce
  volumeMode: Block
  resources:
    requests:
      storage: 10Gi
```

在 Pod 规格配置中为容器添加原始块设备

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-with-block-volume
spec:
  containers:
    - name: fc-container
      image: fedora:26
      command: ["/bin/sh", "-c"]
      args: [ "tail -f /dev/null" ]
      volumeDevices:
        - name: data
          devicePath: /dev/xvda
  volumes:
    - name: data
      persistentVolumeClaim:
        claimName: block-pvc
```

注意：当为 Pod 增加原始块设备时，我们在容器中指定设备路径而不是挂载路径。

绑定块卷

如果用户通过使用 `PersistentVolumeClaim` 规范中的 `volumeMode` 字段指示此请求来请求原始块卷，则绑定规则与以前不认为该模式为规范一部分的版本略有不同。

下面是用户和管理员指定请求原始块设备的可能组合的表格。该表指示卷是否将被绑定或未给定组合。静态设置的卷的卷绑定矩阵：

PV volumeMode	PVC volumeMode	结果
unspecified	unspecified	绑定
unspecified	Block	不绑定
unspecified	Filesystem	绑定
Block	unspecified	不绑定
Block	Block	绑定
Block	Filesystem	不绑定
Filesystem	Filesystem	绑定
Filesystem	Block	不绑定
Filesystem	unspecified	绑定

注意：alpha 版本只支持静态配置卷。使用原始块设备时，管理员应该注意考虑这些值。

编写可移植配置

如果您正在编写在多种群集上运行并需要持久存储的配置模板或示例，我们建议您使用以下模式：

- 不要在您的配置组合中包含 `PersistentVolumeClaim` 对象（与 Deployment、ConfigMap等一起）。
- 不要在配置中包含 `PersistentVolume` 对象，因为用户实例化配置可能没有创建 `PersistentVolume` 的权限。
- 给用户在实例化模板时提供存储类名称的选项。
    - 如果用户提供存储类名称，则将该值放入 `persistentVolumeClaim.storageClassName` 字段中。如果集群具有由管理员启用的 StorageClass，这将导致 PVC 匹配正确的存储类别。
    - 如果用户未提供存储类名称，则将 `persistentVolumeClaim.storageClassName` 字段保留为 nil。
        - 这将导致使用集群中默认的 StorageClass 为用户自动配置 PV。许多集群环境都有默认的 StorageClass，或者管理员可以创建自己的默认 StorageClass。
- 在您的工具中，请注意一段时间之后仍未绑定的 PVC，并向用户展示它们，因为这表示集群可能没有动态存储支持（在这种情况下用户应创建匹配的 PV），或集群没有存储系统（在这种情况下用户不能部署需要 PVC 的配置）。

原文地址：https://kubernetes.iohttps://kubernetes.io/docs/concepts/storage/persistent-volumes/

译者：rootsongjc

StorageClass

本文介绍了 Kubernetes 中 `StorageClass` 的概念。在阅读本文之前建议先熟悉卷 和 Persistent Volume（持久卷）。

介绍

`StorageClass` 为管理员提供了描述存储 "class（类）" 的方法。 不同的 class 可能会映射到不同的服务质量等级或备份策略，或由群集管理员确定的任意策略。 Kubernetes 本身不清楚各种 class 代表的什么。这个概念在其他存储系统中有时被称为“配置文件”。

StorageClass 资源

`StorageClass` 中包含 `provisioner` 、 `parameters` 和 `reclaimPolicy` 字段，当 class 需要动态分配 `PersistentVolume` 时会使用到。

`StorageClass` 对象的名称很重要，用户使用该类来请求一个特定的方法。 当创建 `StorageClass` 对象时，管理员设置名称和其他参数，一旦创建了对象就不能再对其更新。

管理员可以为没有申请绑定到特定 class 的 PVC 指定一个默认的 `StorageClass` ： 更多详情请参阅 `PersistentVolumeClaim` 章节。

```yaml
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: standard
provisioner: kubernetes.io/aws-ebs
parameters:
  type: gp2
reclaimPolicy: Retain
mountOptions:
  - debug
```

Provisioner（存储分配器）

Storage class 有一个分配器，用来决定使用哪个卷插件分配 PV。该字段必须指定。

Volume Plugin	Internal Provisioner	Config Example
AWSElasticBlockStore	✓	AWS
AzureFile	✓	Azure File
AzureDisk	✓	Azure Disk
CephFS	-	-
Cinder	✓	OpenStack Cinder
FC	-	-
FlexVolume	-	-
Flocker	✓	-
GCEPersistentDisk	✓	GCE
Glusterfs	✓	Glusterfs
iSCSI	-	-
PhotonPersistentDisk	✓	-
Quobyte	✓	Quobyte
NFS	-	-
RBD	✓	Ceph RBD
VsphereVolume	✓	vSphere
PortworxVolume	✓	Portworx Volume
ScaleIO	✓	ScaleIO
StorageOS	✓	StorageOS

您不限于指定此处列出的"内置"分配器（其名称前缀为 kubernetes.io 并打包在 Kubernetes 中）。 您还可以运行和指定外部分配器，这些独立的程序遵循由 Kubernetes 定义的 规范。 外部供应商的作者完全可以自由决定他们的代码保存于何处、打包方式、运行方式、使用的插件（包括Flex）等。 代码仓库 kubernetes-incubator/external-storage 包含一个用于为外部分配器编写功能实现的类库，以及各种社区维护的外部分配器。

例如，NFS 没有内部分配器，但可以使用外部分配器。一些外部分配器在代码仓库 kubernetes-incubator/external-storage 中。 也有第三方存储供应商提供自己的外部分配器。

关于内置的 StorageClass 的配置请参考 Storage Classes。

回收策略

由 storage class 动态创建的 Persistent Volume 会在的 `reclaimPolicy` 字段中指定回收策略，可以是 `Delete` 或者 `Retain` 。如果 `StorageClass` 对象被创建时没有指定 `reclaimPolicy` ，它将默认为 `Delete` 。

通过 storage class 手动创建并管理的 Persistent Volume 会使用它们被创建时指定的回收政策。

挂载选项

由 storage class 动态创建的 Persistent Volume 将使用 class 中 `mountOptions` 字段指定的挂载选项。

如果卷插件不支持挂载选项，却指定了该选项，则分配操作失败。 安装选项在 class 和 PV 上都不会做验证，所以如果挂载选项无效，那么这个 PV 就会失败。

参数

Storage class 具有描述属于 storage class 卷的参数。取决于 `分配器` ，可以接受不同的参数。 例如，参数 `type` 的值 `io1` 和参数 `iopsPerGB` 特定于 EBS PV。当参数被省略时，会使用默认值。

参考

- https://kubernetes.io/docs/concepts/storage/storage-classes/

本地持久化存储

本地持久化卷允许用户通过标准 PVC 接口以简单便携的方式访问本地存储。PV 中包含系统用于将 Pod 安排到正确节点的节点亲和性信息。

一旦配置了本地卷，外部静态配置器（provisioner）可用于帮助简化本地存储管理。请注意，本地存储配置器与大多数配置器不同，并且尚不支持动态配置。相反，它要求管理员预先配置每个节点上的本地卷，并且这些卷应该是：

1. Filesystem volumeMode（默认）PV—— 将它们挂载到发现目录下。
2. Block volumeMode PV——在发现目录下为节点上的块设备创建一个符号链接。

配置器将通过为每个卷创建和清除 PersistentVolumes 来管理发现目录下的卷。

配置要求

- 本地卷插件希望路径稳定，包括在重新启动时和添加或删除磁盘时。
- 静态配置器仅发现挂载点（对于文件系统模式卷）或符号链接（对于块模式卷）。对于基于目录的本地卷必须绑定到发现目录中。

版本兼容性

推荐配置器版本与Kubernetes版本

Provisioner version	K8s version	Reason
2.1.0	1.10	Beta API default, block
2.0.0	1.8, 1.9	Mount propagation
1.0.1	1.7	

K8s功能状态

另请参阅已知问题和 CHANGELOG。

1.10：Beta

- 添加了新的 `PV.NodeAffinity` 字段。
- 重要： Alpha PV NodeAffinity annotation 已弃用。用户必须手动更新其 PV 以使用新的 NodeAffinity字段或运行一次性更新作业。
- Alpha：添加了对 raw block 的支持。

1.9：Alpha

- 新的 StorageClass `volumeBindingMode` 参数将延迟PVC绑定，直到 pod 被调度。

1.7：Alpha

- 新的 `local` PersistentVolume 源，允许指定具有 node affinity 的目录或挂载点。
- 使用绑定到该 PV 的 PVC 的 Pod 将始终调度到该节点。

未来的功能

- 本地块设备作为卷源，具有分区和 fs 格式化
- 共享本地持久化存储的动态资源调配
- 当地 PV 健康监测、污点和容忍
- 内联 PV（使用专用本地磁盘作为临时存储）

用户指南

这些说明反映了最新版本的代码库。有关旧版本的说明，请参阅版本兼容性下的版本链接。

步骤1：使用本地磁盘启动集群

启用alpha feature gate

1.10+

如果需要原始的本地块功能，

```
export KUBE_FEATURE_GATES ="BlockVolume = true"
```

注意：1.10 之前的 Kubernetes 版本需要几个附加 feature gate，因为持久的本地卷和其他功能处于 alpha 版本。

选项1：裸金属环境

1. 根据应用程序的要求对每个节点上的磁盘进行分区和格式化。
2. 根据 StorageClass 将所有文件系统挂载到同一个目录下。目录在 configmap 中指定，见下文。
3. 使用 `KUBE_FEATURE_GATES` 配置 Kubernetes API server、controller manager、scheduler 和所有kubelet，如上所述。
4. 如果不使用默认 Kubernetes 调度程序策略，则必须启用以下谓词：
   - 1.9之前： `NoVolumeBindConflict`
   - 1.9+： `VolumeBindingChecker`

选项2：本地测试集群

1. 创建 `/mnt/disks` 目录并将多个卷挂载到其子目录。下面的示例使用三个 ram 磁盘来模拟真实的本地卷：

   ```
   mkdir/mnt/disks
   vol for vol1 vol2 vol3;do
   mkdir/mnt/disks/$vol
   mount -t tmpfs $vol/mnt/disks/$vol
   DONE
   ```

2. 运行本地集群。

   ```
   $ALLOW_PRIVILEGED = true LOG_LEVEL = 5 FEATURE_GATES = $KUBE
   _FEATURE_GATES hack/local-up-cluster.sh
   ```

步骤2：创建StorageClass（1.9+）

要延迟卷绑定，直到 pod 被调度，并在单个 pod 中处理多个本地 PV，必须使用设置为 `WaitForFirstConsumer` 的 `volumeBindingMode` 创建 StorageClass。

```
$kubectl create -f provisioner/deployment/kubernetes/example/def
ault_example_storageclass.yaml
```

步骤3：创建本地持久卷

选项1：使用本地卷静态配置器

1. 生成 Provisioner 的 ServiceAccount、Role、DaemonSet 和 ConfigMap 规范，并对其进行自定义。

   这一步使用 helm 模板来生成规格。有关安装说明，请参阅helm readme。要使用默认值生成配置器的规格，请运行：

   ```
   helm template ./helm/provisioner > ./provisioner/deployment/
   kubernetes/provisioner_generated.yaml
   ```

   您也可以提供一个自定义值文件：

   ```
   helm template ./helm/provisioner --values custom-values.yaml
   > ./provisioner/deployment/kubernetes/provisioner_generated.
   yaml
   ```

2. 部署配置程序

   如果用户对 Provisioner 的 yaml 文件的内容感到满意，可以用 **kubectl** 创建 Provisioner 的 DaemonSet 和 ConfigMap。

   ```
   $kubectl create -f ./provisioner/deployment/kubernetes/provi
   sioner_generated.yaml
   ```

3. 检查发现的本地卷

   一旦启动，外部静态配置器将发现并创建本地 PV。

   例如，如果目录 `/mnt/disks/` 包含一个目录 `/mnt/disks/vol1`，则静态配置器会创建以下本地卷 PV：

```
$ kubectl get pv
NAME                CAPACITY    ACCESSMODES   RECLAIMPOLICY
  STATUS      CLAIM     STORAGECLASS    REASON    AGE
local-pv-ce05be60   1024220Ki   RWO           Delete
  Available             local-storage             26s

$ kubectl describe pv local-pv-ce05be60
Name:          local-pv-ce05be60
Labels:          <none>
Annotations:    pv.kubernetes.io/provisioned-by=local-volume
-provisioner-minikube-18f57fb2-a186-11e7-b543-080027d51893
StorageClass:    local-fast
Status:          Available
Claim:
Reclaim Policy:     Delete
Access Modes:    RWO
Capacity:      1024220Ki
NodeAffinity:
  Required Terms:
      Term 0:  kubernetes.io/hostname in [my-node]
Message:
Source:
    Type:     LocalVolume (a persistent volume backed by loca
l storage on a node)
    Path:     /mnt/disks/vol1
Events:          <none>
```

上面描述的 PV 可以通过引用 `local-fast` storageClassName 声明和绑定到 PVC。

选项2：手动创建本地持久化卷

有关示例 PersistentVolume 规范，请参阅Kubernetes文档。

步骤4：创建本地持久卷声明

```yaml
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: example-local-claim
spec:
  accessModes:
  - ReadWriteOnce
  resources:
    requests:
      storage: 5Gi
  storageClassName: local-storage
```

请替换以下元素以反映您的配置：

- 卷所需的存储容量“5Gi”
- “local-storage”，与本地 PV 关联的存储类名称应该用于满足此 PVC

对于试图声明 “Block” PV 的 “Block” volumeMode PVC，可以使用以下示例：

```yaml
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: example-local-claim
spec:
  accessModes:
  - ReadWriteOnce
  resources:
    requests:
      storage: 5Gi
  volumeMode: Block
  storageClassName: local-storage
```

请注意，此处唯一需要注意的字段是 volumeMode，它已被设置为“Block”。

最佳实践

- 对于IO隔离，建议每个卷使用整个磁盘
- 对于容量隔离，建议使用单个分区
- 避免重新创建具有相同节点名称的节点，而仍然存在指定了该节点亲和性的旧 PV。否则，系统可能认为新节点包含旧的 PV。
- 对于带有文件系统的卷，建议在 fstab 条目和该挂载点的目录名称中使用它们的 UUID（例如 `ls -l/dev/disk/by-uuid` 的输出）。这种做法可确保即使设备路径发生变化（例如，如果 `/dev/sda1` 在添加新磁盘时变为

`/dev/sdb1` ），也不会错误地挂在本地卷。此外，这种做法将确保如果创建具有相同名称的另一个节点，则该节点上的任何卷都是唯一的，而不会误认为是具有相同名称的另一个节点上的卷。

- 对于没有文件系统的 raw block 卷，使用唯一的 ID 作为符号链接名称。根据您的环境，`/dev/disk/by-id/` 中的卷 ID 可能包含唯一的硬件序列号。否则，应该生成一个唯一的 ID。符号链接名称的唯一性将确保如果创建具有相同名称的另一个节点，则该节点上的任何卷都是唯一的，而不会误认为是具有相同名称的另一个节点上的卷。

删除/清理底层卷

当您想要停用本地卷时，以下是可能的工作流程。

1. 停止使用卷的 pod
2. 从节点中删除本地卷（即卸载、拔出磁盘等）
3. 删除 PVC
4. 供应商将尝试清理卷，但由于卷不再存在而会失败
5. 手动删除 PV 对象

参考

- Local Persistent Storage User Guide

扩展

Kubernetes是一个高度开放可扩展的架构，可以通过自定义资源类型CRD来定义自己的类型，还可以自己来扩展API服务，用户的使用方式跟Kubernetes的原生对象无异。

使用自定义资源扩展API

> 注意：TPR已经停止维护，kubernetes 1.7及以上版本请使用CRD。

自定义资源是对Kubernetes API的扩展，kubernetes中的每个资源都是一个API对象的集合，例如我们在YAML文件里定义的那些spec都是对kubernetes中的资源对象的定义，所有的自定义资源可以跟kubernetes中内建的资源一样使用kubectl操作。

自定义资源

Kubernetes1.6版本中包含一个内建的资源叫做TPR（ThirdPartyResource），可以用它来创建自定义资源，但该资源在kubernetes1.7中版本已被CRD（CustomResourceDefinition）取代。

扩展API

自定义资源实际上是为了扩展kubernetes的API，向kubenetes API中增加新类型，可以使用以下三种方式：

- 修改kubenetes的源码，显然难度比较高，也不太合适
- 创建自定义API server并聚合到API中
- 1.7以下版本编写TPR，kubernetes1.7及以上版本用CRD

编写自定义资源是扩展kubernetes API的最简单的方式，是否编写自定义资源来扩展API请参考Should I add a custom resource to my Kubernetes Cluster?，行动前请先考虑以下几点：

- 你的API是否属于声明式的
- 是否想使用kubectl命令来管理
- 是否要作为kubenretes中的对象类型来管理，同时显示在kuberetes dashboard上
- 是否可以遵守kubernetes的API规则限制，例如URL和API group、namespace限制
- 是否可以接受该API只能作用于集群或者namespace范围

- 想要复用kubernetes API的公共功能，比如CRUD、watch、内置的认证和授权等

如果这些都不是你想要的，那么你可以开发一个独立的API。

TPR

> 注意：TPR已经停止维护，kubernetes 1.7及以上版本请使用CRD。

假如我们要创建一个名为 `cron-tab.stable.example.com` 的TPR，yaml文件定义如下：

```yaml
apiVersion: extensions/v1beta1
kind: ThirdPartyResource
metadata:
  name: cron-tab.stable.example.com
description: "A specification of a Pod to run on a cron style schedule"
versions:
- name: v1
```

然后使用 `kubectl create` 命令创建该资源，这样就可以创建出一个API端点 `/apis/stable.example.com/v1/namespaces/<namespace>/crontabs/...` 。

下面是在Linkerd中的一个实际应用，Linkerd中的一个名为namerd的组件使用了TPR，定义如下：

```yaml
---
kind: ThirdPartyResource
apiVersion: extensions/v1beta1
metadata:
  name: d-tab.l5d.io
description: stores dtabs used by namerd
versions:
- name: v1alpha1
```

CRD

参考下面的CRD，resourcedefinition.yaml：

```yaml
apiVersion: apiextensions.k8s.io/v1beta1
kind: CustomResourceDefinition
metadata:
  # 名称必须符合下面的格式：<plural>.<group>
  name: crontabs.stable.example.com
spec:
  # REST API使用的组名称：/apis/<group>/<version>
  group: stable.example.com
  # REST API使用的版本号：/apis/<group>/<version>
  version: v1
  # Namespaced或Cluster
  scope: Namespaced
  names:
    # URL中使用的复数名称: /apis/<group>/<version>/<plural>
    plural: crontabs
    # CLI中使用的单数名称
    singular: crontab
    # CamelCased格式的单数类型。在清单文件中使用
    kind: CronTab
    # CLI中使用的资源简称
    shortNames:
    - ct
```

创建该CRD：

```
kubectl create -f resourcedefinition.yaml
```

访问RESTful API端点如http://172.20.0.113:8080将看到如下API端点已创建：

```
/apis/stable.example.com/v1/namespaces/*/crontabs/...
```

创建自定义对象

如下所示：

```yaml
apiVersion: "stable.example.com/v1"
kind: CronTab
metadata:
  name: my-new-cron-object
spec:
  cronSpec: "* * * * /5"
  image: my-awesome-cron-image
```

引用该自定义资源的API创建对象。

终止器

可以为自定义对象添加一个终止器，如下所示：

```
apiVersion: "stable.example.com/v1"
kind: CronTab
metadata:
  finalizers:
  - finalizer.stable.example.com
```

删除自定义对象前，异步执行的钩子。对于具有终止器的一个对象，删除请求仅仅是为 `metadata.deletionTimestamp` 字段设置一个值，而不是删除它，这将触发监控该对象的控制器执行他们所能处理的任意终止器。

详情参考：Extend the Kubernetes API with CustomResourceDefinitions

使用kubernetes1.7及以上版本请参考Migrate a ThirdPartyResource to CustomResourceDefinition。

自定义控制器

单纯设置了自定义资源，并没有什么用，只有跟自定义控制器结合起来，才能讲资源对象中的声明式API翻译成用户所期望的状态。自定义控制器可以用来管理任何资源类型，但是一般是跟自定义资源结合使用。

请参考使用Operator模式，该模式可以让开发者将自己的领域知识转换成特定的kubenretes API扩展。

API server聚合

Aggregated（聚合的）API server是为了将原来的API server这个巨石（monolithic）应用给拆分成，为了方便用户开发自己的API server集成进来，而不用直接修改kubernetes官方仓库的代码，这样一来也能将API server解耦，方便用户使用实验特性。这些API server可以跟core API server无缝衔接，试用kubectl也可以管理它们。

详情参考Aggregated API Server。

参考

- Custom Resources
- Extend the Kubernetes API with CustomResourceDefinitions
- Introducing Operators: Putting Operational Knowledge into Software

Aggregated API Server

Aggregated（聚合的）API server是为了将原来的API server这个巨石（monolithic）应用给拆分成，为了方便用户开发自己的API server集成进来，而不用直接修改kubernetes官方仓库的代码，这样一来也能将API server解耦，方便用户使用实验特性。这些API server可以跟core API server无缝衔接，使用kubectl也可以管理它们。

架构

我们需要创建一个新的组件，名为 `kube-aggregator` ，它需要负责以下几件事：

- 提供用于注册API server的API
- 汇总所有的API server信息
- 代理所有的客户端到API server的请求

注意：这里说的API server是一组“API Server”，而不是说我们安装集群时候的那个API server，而且这组API server是可以横向扩展的。

关于聚合的API server的更多信息请参考：Aggregated API Server

安装配置聚合的API server

有两种方式来启用 `kube-aggregator` ：

- 使用**test mode/single-user mode**，作为一个独立的进程来运行
- 使用**gateway mode**， `kube-apiserver` 将嵌入到 `kbe-aggregator` 组件中，它将作为一个集群的gateway，用来聚合所有apiserver。

`kube-aggregator` 二进制文件已经包含在kubernetes release里面了。

参考

Aggregated API Servers - kuberentes design-proposals

APIService

APIService是用来表示一个特定的 `GroupVersion` 的中的server，它的结构定义位于代码 `staging/src/k8s.io/kube-aggregator/pkg/apis/apiregistration/types.go` 中。

下面是一个APIService的示例配置：

```yaml
apiVersion: apiregistration.k8s.io/v1beta1
kind: APIService
metadata:
  name: v1alpha1.custom-metrics.metrics.k8s.io
spec:
  insecureSkipTLSVerify: true
  group: custom-metrics.metrics.k8s.io
  groupPriorityMinimum: 1000
  versionPriority: 5
  service:
    name: api
    namespace: custom-metrics
  version: v1alpha1
```

APIService详解

使用 `apiregistration.k8s.io/v1beta1` 版本的APIService，在metadata.name中定义该API的名字。

使用上面的yaml的创建 `v1alpha1.custom-metrics.metrics.k8s.io` APIService。

- `insecureSkipTLSVerify` ：当与该服务通信时，禁用TLS证书认证。强加建议不要设置这个参数，默认为 false。应该使用CABundle代替。
- `service` ：与该APIService通信时引用的service，其中要注明service的名字和所属的namespace，如果为空的话，则所有的服务都会该API groupversion将在本地443端口处理所有通信。
- `groupPriorityMinimum` ：该组API的处理优先级，主要排序是基于 `groupPriorityMinimum` ，该数字越大表明优先级越高，客户端就会与其通信处理请求。次要排序是基于字母表顺序，例如v1.bar比v1.foo的优先级更高。

- `versionPriority`：VersionPriority控制其组内的API版本的顺序。必须大于零。主要排序基于VersionPriority，从最高到最低（20大于10）排序。次要排序是基于对象名称的字母比较。（v1.foo在v1.bar之前）由于它们都是在一个组内，因此数字可能很小，一般都小于10。

查看我们使用上面的yaml文件创建的APIService。

```
kubectl get apiservice v1alpha1.custom-metrics.metrics.k8s.io -o
 yaml
```


```yaml
apiVersion: apiregistration.k8s.io/v1beta1
kind: APIService
metadata:
  creationTimestamp: 2017-12-14T08:27:35Z
  name: v1alpha1.custom-metrics.metrics.k8s.io
  resourceVersion: "35194598"
  selfLink: /apis/apiregistration.k8s.io/v1beta1/apiservices/v1a
lpha1.custom-metrics.metrics.k8s.io
  uid: a31a3412-e0a8-11e7-9fa4-f4e9d49f8ed0
spec:
  caBundle: null
  group: custom-metrics.metrics.k8s.io
  groupPriorityMinimum: 1000
  insecureSkipTLSVerify: true
  service:
    name: api
    namespace: custom-metrics
  version: v1alpha1
  versionPriority: 5
status:
  conditions:
  - lastTransitionTime: 2017-12-14T08:27:38Z
    message: all checks passed
    reason: Passed
    status: "True"
    type: Available
```


查看集群支持的APISerivce

作为Kubernetes中的一种资源对象，可以使用 `kubectl get apiservice` 来查看。

例如查看集群中所有的APIService：

```
$ kubectl get apiservice
NAME                                    AGE
v1.                                     2d
v1.authentication.k8s.io                2d
v1.authorization.k8s.io                 2d
v1.autoscaling                          2d
v1.batch                                2d
v1.monitoring.coreos.com                1d
v1.networking.k8s.io                    2d
v1.rbac.authorization.k8s.io            2d
v1.storage.k8s.io                       2d
v1alpha1.custom-metrics.metrics.k8s.io  2h
v1beta1.apiextensions.k8s.io            2d
v1beta1.apps                            2d
v1beta1.authentication.k8s.io           2d
v1beta1.authorization.k8s.io            2d
v1beta1.batch                           2d
v1beta1.certificates.k8s.io             2d
v1beta1.extensions                      2d
v1beta1.policy                          2d
v1beta1.rbac.authorization.k8s.io       2d
v1beta1.storage.k8s.io                  2d
v1beta2.apps                            2d
v2beta1.autoscaling                     2d
```

另外查看当前kubernetes集群支持的API版本还可以使用 `kubectl api-version` ：

```
$ kubectl api-versions
apiextensions.k8s.io/v1beta1
apiregistration.k8s.io/v1beta1
apps/v1beta1
apps/v1beta2
authentication.k8s.io/v1
authentication.k8s.io/v1beta1
authorization.k8s.io/v1
authorization.k8s.io/v1beta1
autoscaling/v1
autoscaling/v2beta1
batch/v1
batch/v1beta1
certificates.k8s.io/v1beta1
custom-metrics.metrics.k8s.io/v1alpha1
extensions/v1beta1
monitoring.coreos.com/v1
networking.k8s.io/v1
policy/v1beta1
rbac.authorization.k8s.io/v1
rbac.authorization.k8s.io/v1beta1
storage.k8s.io/v1
storage.k8s.io/v1beta1
v1
```

参考

API Conventions

Kuberentes1.8 reference doc

服务目录（Service Catalog）

服务目录（Service Catalog）是Kubernetes的扩展API，它使运行在Kubernetes集群中的应用程序可以轻松使用外部托管软件产品，例如由云提供商提供的数据存储服务。

它提供列表清单、提供(provision)和绑定(binding)来自服务代理（Service Brokers）的外部托管服务，而不需要关心如何创建或管理这些服务的详细情况。

由Open Service Broker API规范定义的Service broker是由第三方提供和维护的一组托管服务的端点(endpoint)，该第三方可以是AWS，GCP或Azure等云提供商。

托管服务可以是Microsoft Azure Cloud Queue，Amazon Simple Queue Service和Google Cloud Pub/Sub等，它们可以是应用可以使用的提供的各种软件。

通过Service Catalog，集群运营者可以浏览由Service Broker提供的托管服务列表，提供的托管服务实例，并与其绑定，使其可被Kubernetes集群中的应用程序所使用。

场景样例

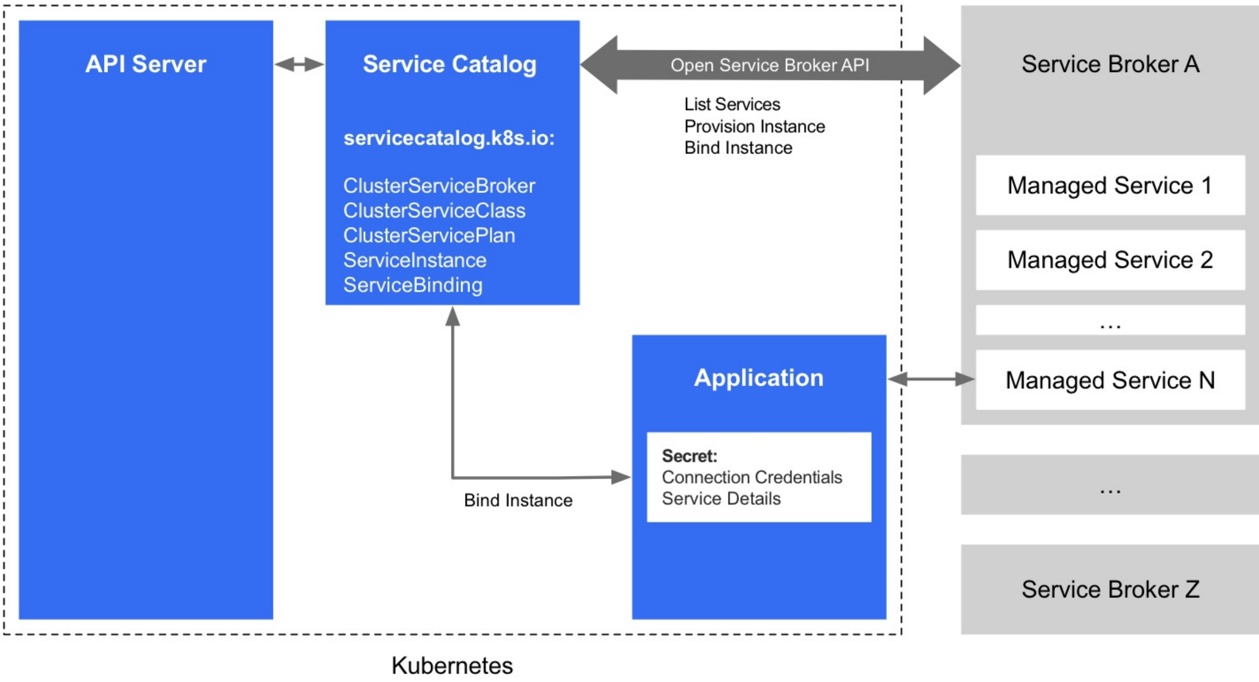
应用程序开发人员编写基于Kubernetes集群的应用程序，他们希望使用消息队列作为其应用程序的一部分。但是，他们不想自己配置和管理这个服务服务。恰好，有一家云提供商通过其服务代理(Service Broker)提供消息队列服务。

集群运营商可以设置Service Catalog并使用它与云提供商的Service Broker进行通信，以调配消息排队服务的实例并使其可用于Kubernetes集群内的应用程序。因此，应用程序开发人员不需要关心消息队列的实现细节或管理，可以简单地像服务一样使用它。

架构

Service Catalog使用Open Service Broker API与Service Broker进行通信，充当Kubernetes API服务器的中介，发起供应并返回应用程序使用托管服务所需的凭据。

Service Catalog通过扩展API服务器和控制器实现，使用etcd进行存储。它还使用Kubernetes 1.7+中提供的聚合层来呈现其API。



图片 - *Service Catalog Architecture*

API资源

Service Catalog安装servicecatalog.k8s.ioAPI并提供以以下Kubernetes资源：

- ClusterServiceBroker：作为service broker的群集内代理，封装其服务器连接详细信息。这些由集群运营者创建和管理，希望使用broker服务在其集群中提供新类型的托管服务。

- ClusterServiceClass：由特定service broker提供的托管服务。将新ClusterServiceBroker资源添加到群集时，Service catalog controller将连接到service broker以获取可用托管服务的列表清单。然后它会创建新的ClusterServiceClass资源，与每个托管服务相对应。

- ClusterServicePlan：托管服务的特定产品。例如，托管服务可能有不同的可用套餐，例如免费套餐或付费套餐，或者可能有不同的配置选项，例如使用SSD存储或拥有更多资源。同向群集添加ClusterServiceClass一样，当添加一个新的ClusterServiceBroker时，Service Catalog会创建一个新的ClusterServicePlan资源，与每个托管服务可用的每个服务套餐对应。

- ServiceInstance：一个提供好的ClusterServiceClass实例。这些是由集群运营者创建的托管服务的特定实例，供一个或多个集群内应用程序使用。当创建一个新的ServiceInstance资源时，Service Catalog controller连接到相应的服务代理并指示它提供服务实例。
- ServiceBinding：访问ServiceInstance的凭据。由想让他们的应用利用ServiceInstance的集群集运营者创建。创建之后，Service Catalog controller将创建一个与此服务实例对应的Kubernetes的Secret，包含此服务实例的连接详细信息和凭证 ，可以挂载到Pod中。

鉴权认证

Service Catalog 支持这些认证方法：

- Basic (username/password)
- OAuth 2.0 Bearer Token

用法

群集运营者可以使用Service Catalog API资源来提供托管服务，并使其在Kubernetes群集中可用。涉及的步骤是：

1. 列出Service Broker提供的托管服务清单和服务套餐。
2. 提供托管服务的新实例。
3. 绑定到托管服务，该服务返回连接凭证。
4. 将连接凭证映射到应用程序中。

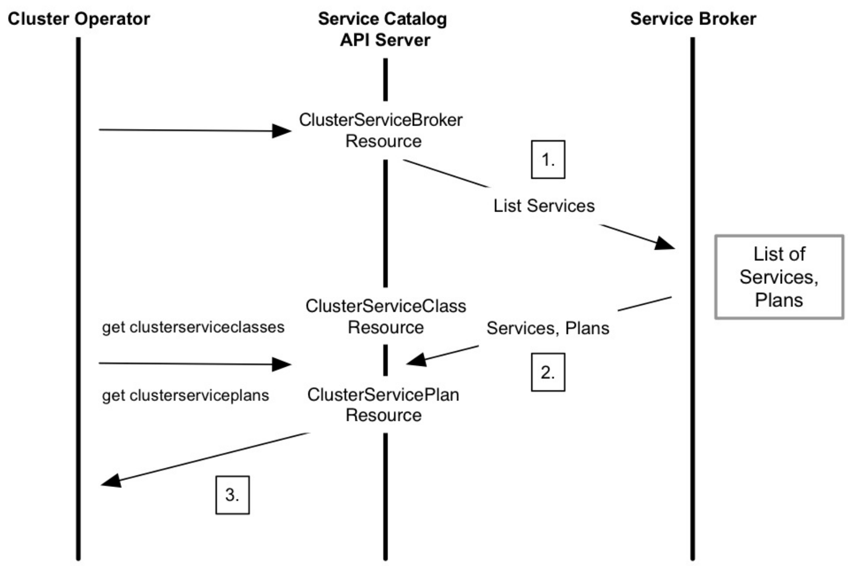
列出托管服务和服务套餐

首先，群集运营者必须在servicecatalog.k8s.io群组内创建ClusterServiceBroker资源。此资源包含访问服务代理端点所需的URL和连接详细信息。

这是一个ClusterServiceBroker资源的例子：

```yaml
apiVersion: servicecatalog.k8s.io/v1beta1
kind: ClusterServiceBroker
metadata:
  name: cloud-broker
spec:
  # Points to the endpoint of a service broker. (This example is
 not a working URL.)
  url:  https://servicebroker.somecloudprovider.com/v1alpha1/pro
jects/service-catalog/brokers/default
  #####
  # Additional values can be added here, which may be used to co
mmunicate
  # with the service broker, such as bearer token info or a caBu
ndle for TLS.
  #####
```

以下是说明从一个service broker列出托管服务和套餐所涉及步骤的顺序图：



图片 - *List Services*

1. 将ClusterServiceBroker资源添加到Service catalog中，它会触发对外部Service Broker的调用以获取可用服务的清单。
2. Service Broker返回可用托管服务的清单和服务套餐的列表，它们分别在本地缓存为 `ClusterServiceClass` 资源和 `ClusterServicePlan` 资源。
3. 然后，集群运营者可以使用以下命令获取可用托管服务的清单：

```
kubectl get clusterserviceclasses -o=custom-columns=SERVICE\ NAME:.metadata.name,EXTERNAL\ NAME:.spec.externalName
```

它应该输出一个类似于以下格式的服务名称列表：

```
SERVICE NAME                            EXTERNAL NAME
4f6e6cf6-ffdd-425f-a2c7-3c9258ad2468    cloud-provider-service
...                                     ...
```

他们还可以使用以下命令查看可用的服务套餐：

```
kubectl get clusterserviceplans -o=custom-columns=PLAN\ NAME:.metadata.name,EXTERNAL\ NAME:.spec.externalName
```

它应该输出一个类似于以下格式的套餐名称列表：

```
PLAN NAME                               EXTERNAL NAME
86064792-7ea2-467b-af93-ac9694d96d52    service-plan-name
...                                     ...
```

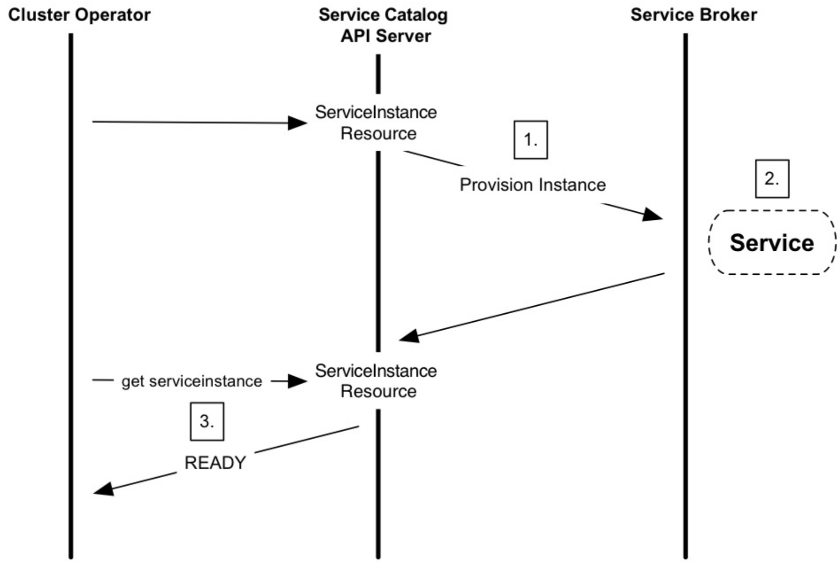
提供新的实例

集群运营者可以通过创建ServiceInstance资源来启动新实例的供应。

如下是一个ServiceInstance资源的例子：

```yaml
apiVersion: servicecatalog.k8s.io/v1beta1
kind: ServiceInstance
metadata:
  name: cloud-queue-instance
  namespace: cloud-apps
spec:
  # References one of the previously returned services
  clusterServiceClassExternalName: cloud-provider-service
  clusterServicePlanExternalName: service-plan-name
  #####
  # Additional parameters can be added here,
  # which may be used by the service broker.
  #####
```

以下序列图说明了提供一个新的托管服务的实例所涉及的步骤：



图片 - *Provision a Service*

1. 当 `ServiceInstance` 资源创建后，Service Catalog发起到外部service broker来提供服务的一个实例。
2. service broker创建托管服务的新实例并返回HTTP响应。
3. 然后，群集运营者可以检查实例的状态，来确认它是否准备就绪。

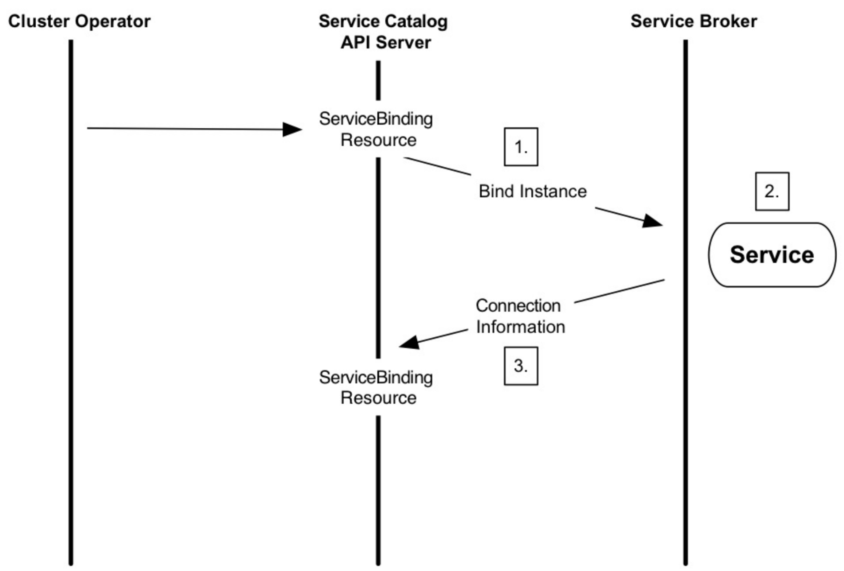
绑定到托管服务

在提供新实例后，群集运营者必须绑定到托管服务才能获取到应用程序使用服务所需的连接凭证和服务帐户详细信息。这是通过创建 `ServiceBinding` 资源完成的。

以下是一个 `ServiceBinding` 资源的例子：

```
apiVersion: servicecatalog.k8s.io/v1beta1
kind: ServiceBinding
metadata:
  name: cloud-queue-binding
  namespace: cloud-apps
spec:
  instanceRef:
    name: cloud-queue-instance
  #####
  # Additional information can be added here, such as a secretNa
me or
  # service account parameters, which may be used by the service
 broker.
  #####
```

以下序列图说明了绑定到托管服务实例所涉及的步骤：

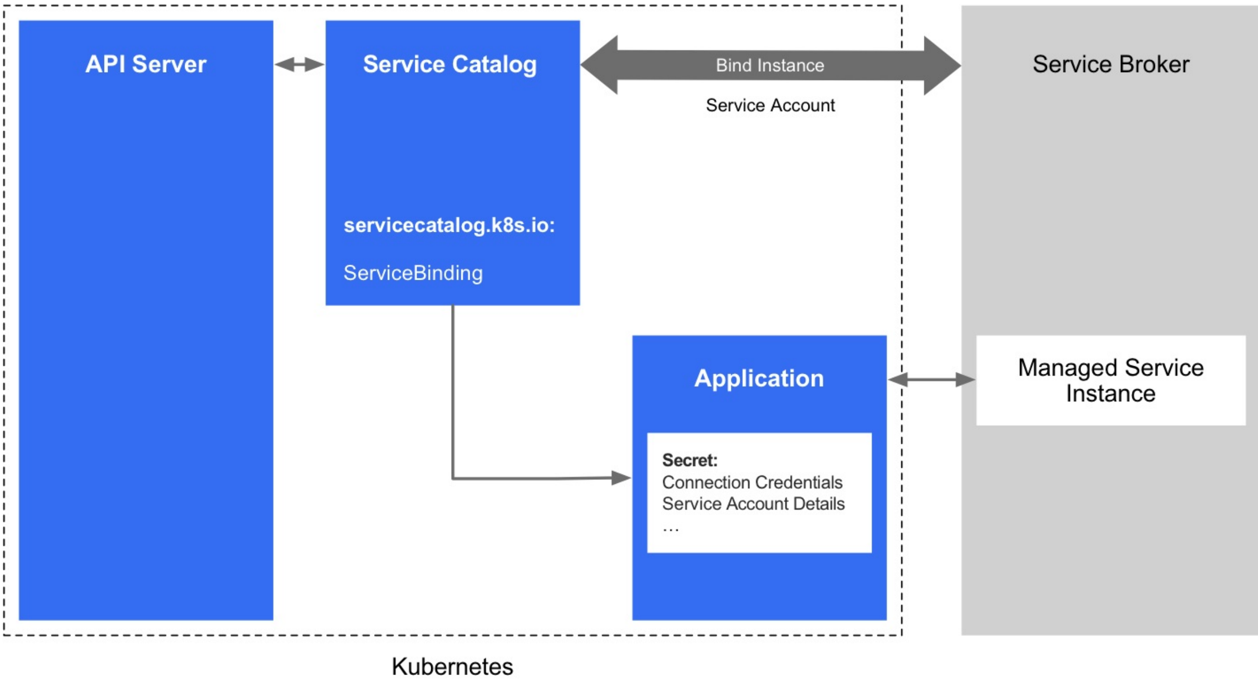


图片 - *Bind to a managed service*

1. 在ServiceBinding创建后，Service Catalog给外部service broker发一个调用请求，获取与服务实例绑定所需的信息。
2. service broker为相应的服务帐户启用应用程序权限/角色。
3. service broker返回连接和访问托管服务实例所需的信息。根据不同的提供商和不同的服务，返回的信息可能在服务提供商和其管理服务之间有所不同。

映射连接凭证

绑定后，最后一步是将连接凭证和服务特定的信息映射到应用程序中。这些信息存储在secret中，应用程序可以用来访问并与托管服务连接。



图片 - *Map connection credentials*

Pod配置文件

执行此映射的一种方法是使用声明式Pod配置文件。

以下示例描述了如何将服务帐户凭证映射到应用程序中。被调用的sa-key密钥存储在名为provider-cloud-key的卷中，并且应用程序将此卷挂载到/var/secrets/provider/key.json。环境变量PROVIDER_APPLICATION_CREDENTIALS是从挂载文件的值映射而来的。

```
...
    spec:
      volumes:
        - name: provider-cloud-key
          secret:
            secretName: sa-key
      containers:
...
          volumeMounts:
          - name: provider-cloud-key
            mountPath: /var/secrets/provider
          env:
          - name: PROVIDER_APPLICATION_CREDENTIALS
            value: "/var/secrets/provider/key.json"
```

以下示例描述如何将secret值映射到应用程序环境变量。在此示例中，消息传递队列 `topic` 名称从名为 `provider-queue-credentials` 的secret的key topic值映射到环境变量 `TOPIC` 。

```
...
          env:
          - name: "TOPIC"
            valueFrom:
                secretKeyRef:
                   name: provider-queue-credentials
                   key: topic
```

下一步

- 如果熟悉Helm Charts ，使用Helm将Service Catalog安装到Kubernetes集群中。或者，可以使用SC工具安装服务目录。
- 查看 sample service brokers.
- 探索kubernetes-incubator/service-catalog 项目。

以上翻译自官方文档。

Service Catalog的安装(利用Helm)和交互

以下翻译自官方项目文档。与官方网站文档大致一致。

Kubernetes 1.7或更高版本的集群运行 API Aggregator，它位于core API Server前面的专用proxy服务器。

服务目录(Service Catalog)提供了一个位于API aggregator后面的API Server，因此可以用kubectl像平常一样与Service Catalog进行交互。

要了解更多关于API aggregation的信息，请参阅 Kubernetes文档。

本文档的其余部分详细介绍了如何：

- 在群集上设置Service Catalog
- 与Service Catalog API进行交互

前提条件

Kubernetes版本

Service Catalog需要Kubernetes v1.7或更高版本。您还需要 在主机上安装 Kubernetes configuration file 。你需要这个文件，以便可以使用kubectl和 helm与群集通信。许多Kubernetes安装工具或云提供商会为你设置此配置文件。有关详细信息，请与您的工具或提供商联系。

`kubectl` 版本

大多数与Service Catalog系统的交互都是通过 `kubectl` 命令行界面实现的。与群集版本一样，Service Catalog需要kubectl版本1.7或更高版本。

首先，检查 `kubectl` 版本：

```
kubectl version
```

确保Kubernetes版本和kubectl版本均为1.7或更高。

如果需要升级客户端，请按照安装说明 获取新的 `kubectl` 二进制文件。

例如，运行以下命令以在Mac OS上获取最新的二进制文件：

```
curl -LO https://storage.googleapis.com/kubernetes-release/release/$(curl -s https://storage.googleapis.com/kubernetes-release/release/stable.txt)/bin/darwin/amd64/kubectl
chmod +x ./kubectl
```

群集内DNS

您需要启用Kubernetes集群内的DNS。大多数常用的安装方法会为您自动配置群集内DNS：

- Minikube
- `hack/local-up-cluster.sh`
- 大多数云提供商

Helm

使用Helm安装Service Catalog ，需要v2.7.0或更高版本。请参阅以下步骤进行安装。

如果还没有安装Helm

如果尚未安装Helm，请下载 `helm` CLI，然后运行helm init（这会将Helm的服务器端组件Tiller安装到Kubernetes群集中）。

如果已经安装了Helm

如果已经安装了Helm，请运行helm version并确保客户端和服务器版本均为v2.7.0或更高。

如果不是， 请安装更新版本的helm CLI并运行 `helm init --upgrade` 。

有关安装的更多详细信息，请参阅 Helm安装说明。

Tiller 权限

Tiller是Helm的服务端组件。默认情况下， helm init将Tiller pod安装到kube-system名称空间中，并将Tiller配置为使用default服务帐户（service account）。

需要对Tiller进行配置 `cluster-admin` 权限，才能正确安装Service Catalog：

```
kubectl create clusterrolebinding tiller-cluster-admin \
    --clusterrole=cluster-admin \
    --serviceaccount=kube-system:default
```

Helm Repository设置

Service Catalog很容易通过Helm chart安装 。

此chart位于 chart repository中。将此repository添加到本地计算机：

```
helm repo add svc-cat https://svc-catalog-charts.storage.googlea
pis.com
```

然后，确保repository已成功添加：

```
helm search service-catalog
```

应该看到以下输出：

```
NAME                VERSION    DESCRIPTION
svc-cat/catalog     x,y.z      service-catalog API server and con
troller-manag...
```

RBAC

Kubernetes群集必须启用RBAC 才能使用Service Catalog。

与群集内DNS一样，许多安装方法都有对应启用RBAC的途径。

Minikube

如果您正在使用Minikube，请使用以下命令启动群集：

```
minikube start --extra-config=apiserver.Authorization.Mode=RBAC
```

hack/local-cluster-up.sh

如果使用 `hack/local-up-cluster.sh` 脚本，请使用以下命令启动群集：

```
AUTHORIZATION_MODE=Node,RBAC hack/local-up-cluster.sh -O
```

云提供商

许多云提供商为你启用了RBAC的新集群。有关详细信息，请查阅你的提供商的文档。

安装Service Catalog

集群和Helm配置正确，安装Service Catalog很简单：

```
helm install svc-cat/catalog \
    --name catalog --namespace catalog
```

安装Service Catalog CLI 客户端

按照适用于操作系统的说明安装svcat。二进制文件可以单独使用，也可以作为kubectl插件使用。

MacOS

```
curl -sLO https://download.svcat.sh/cli/latest/darwin/amd64/svca
t
chmod +x ./svcat
mv ./svcat /usr/local/bin/
svcat version --client
```

Linux

```
curl -sLO https://download.svcat.sh/cli/latest/linux/amd64/svcat
chmod +x ./svcat
mv ./svcat /usr/local/bin/
svcat version --client
```

Windows

下面的片段仅在当前会话的PATH添加一个路径。后续使用需要将它相应的路径永久添加到PATH中。

```
iwr 'https://download.svcat.sh/cli/latest/windows/amd64/svcat.ex
e' -UseBasicParsing -OutFile svcat.exe
mkdir -f ~\bin
$env:PATH += ";${pwd}\bin"
svcat version --client
```

手动方式

1. 对应操作系统下载相应的二进制文件：
    - macOS: https://download.svcat.sh/cli/latest/darwin/amd64/svcat
    - Windows: https://download.svcat.sh/cli/latest/windows/amd64/svcat.exe
    - Linux: https://download.svcat.sh/cli/latest/linux/amd64/svcat
2. 使二进制文件可执行。
3. 将二进制文件移动到PATH相应的目录。

插件方式使用客户端

要将svcat用作插件，请在下载后运行以下命令：

```
$ ./svcat install plugin
Plugin has been installed to ~/.kube/plugins/svcat. Run kubectl
plugin svcat --help for help using the plugin.
```

当作为插件运行时，这些命令与添加全局kubectl配置标志相同。其中一个例外是，在插件模式下运行时不支持布尔标志，所以不要使用 `--flag` ,必须指定 `--flag=true` 。

资源调度

Kubernetes作为一个容器编排调度引擎，资源调度是它的最基本也是最重要的功能，这一节中我们将着重讲解Kubernetes中是如何做资源调度的。

Kubernetes中有一个叫做 `kube-scheduler` 的组件，该组件就是专门监听 `kube-apiserver` 中是否有还未调度到node上的pod，再通过特定的算法为pod指定分派node运行。

Kubernetes中的众多资源类型，例如Deployment、DaemonSet、StatefulSet等都已经定义了Pod运行的一些默认调度策略，但是如果我们细心的根据node或者pod的不同属性，分别为它们打上标签之后，我们将发现Kubernetes中的高级调度策略是多么强大。当然如果要实现动态的资源调度，即pod已经调度到某些节点上后，因为一些其它原因，想要让pod重新调度到其它节点。

考虑以下两种情况：

- 集群中有新增节点，想要让集群中的节点的资源利用率比较均衡一些，想要将一些高负载的节点上的pod驱逐到新增节点上，这是kuberentes的scheduer所不支持的，需要使用如rescheduler这样的插件来实现。
- 想要运行一些大数据应用，设计到资源分片，pod需要与数据分布达到一致均衡，避免个别节点处理大量数据，而其它节点闲置导致整个作业延迟，这时候可以考虑使用kube-arbitritor。

用户指南

该章节主要记录kubernetes使用过程中的一些配置技巧和操作。

- 配置Pod的liveness和readiness探针
- 管理集群中的TLS

资源配置

Kubernetes 中的各个 Object 的配置指南。

配置Pod的liveness和readiness探针

当你使用kuberentes的时候，有没有遇到过Pod在启动后一会就挂掉然后又重新启动这样的恶性循环？你有没有想过kubernetes是如何检测pod是否还存活？虽然容器已经启动，但是kubernetes如何知道容器的进程是否准备好对外提供服务了呢？让我们通过kuberentes官网的这篇文章Configure Liveness and Readiness Probes，来一探究竟。

本文将向展示如何配置容器的存活和可读性探针。

Kubelet使用liveness probe（存活探针）来确定何时重启容器。例如，当应用程序处于运行状态但无法做进一步操作，liveness探针将捕获到deadlock，重启处于该状态下的容器，使应用程序在存在bug的情况下依然能够继续运行下去（谁的程序还没几个bug呢）。

Kubelet使用readiness probe（就绪探针）来确定容器是否已经就绪可以接受流量。只有当Pod中的容器都处于就绪状态时kubelet才会认定该Pod处于就绪状态。该信号的作用是控制哪些Pod应该作为service的后端。如果Pod处于非就绪状态，那么它们将会被从service的load balancer中移除。

定义 liveness命令

许多长时间运行的应用程序最终会转换到broken状态，除非重新启动，否则无法恢复。Kubernetes提供了liveness probe来检测和补救这种情况。

在本次练习将基于 `gcr.io/google_containers/busybox` 镜像创建运行一个容器的Pod。以下是Pod的配置文件 `exec-liveness.yaml` ：

```yaml
apiVersion: v1
kind: Pod
metadata:
  labels:
    test: liveness
  name: liveness-exec
spec:
  containers:
  - name: liveness
    args:
    - /bin/sh
    - -c
    - touch /tmp/healthy; sleep 30; rm -rf /tmp/healthy; sleep 6
00
    image: gcr.io/google_containers/busybox
    livenessProbe:
      exec:
        command:
        - cat
        - /tmp/healthy
      initialDelaySeconds: 5
      periodSeconds: 5
```

该配置文件给Pod配置了一个容器。 `periodSeconds` 规定kubelet要每隔5秒执行一次liveness probe。 `initialDelaySeconds` 告诉kubelet在第一次执行probe之前要的等待5秒钟。探针检测命令是在容器中执行 `cat /tmp/healthy` 命令。如果命令执行成功，将返回0，kubelet就会认为该容器是活着的并且很健康。如果返回非0值，kubelet就会杀掉这个容器并重启它。

容器启动时，执行该命令：

```
/bin/sh -c "touch /tmp/healthy; sleep 30; rm -rf /tmp/healthy; s
leep 600"
```

在容器生命的最初30秒内有一个 `/tmp/healthy` 文件，在这30秒内 `cat /tmp/healthy` 命令会返回一个成功的返回码。30秒后， `cat /tmp/healthy` 将返回失败的返回码。

创建Pod：

```
kubectl create -f https://k8s.io/docs/tasks/configure-pod-contai
ner/exec-liveness.yaml
```

在30秒内，查看Pod的event：

```
kubectl describe pod liveness-exec
```

结果显示没有失败的liveness probe：

```
FirstSeen     LastSeen    Count   From             SubobjectPath
         Type         Reason      Message
--------- --------    -----   ----            -------------
      --------    ------      -------
24s       24s     1   {default-scheduler }                     No
rmal      Scheduled   Successfully assigned liveness-exec to wor
ker0
23s       23s     1   {kubelet worker0}   spec.containers{livene
ss}   Normal      Pulling     pulling image "gcr.io/google_conta
iners/busybox"
23s       23s     1   {kubelet worker0}   spec.containers{livene
ss}   Normal      Pulled      Successfully pulled image "gcr.io/
google_containers/busybox"
23s       23s     1   {kubelet worker0}   spec.containers{livene
ss}   Normal      Created     Created container with docker id 8
6849c15382e; Security:[seccomp=unconfined]
23s       23s     1   {kubelet worker0}   spec.containers{livene
ss}   Normal      Started     Started container with docker id 8
6849c15382e
```

启动35秒后，再次查看pod的event：

```
kubectl describe pod liveness-exec
```

在最下面有一条信息显示liveness probe失败，容器被删掉并重新创建。

```
FirstSeen LastSeen    Count   From            SubobjectPath
      Type        Reason      Message
--------- --------    -----   ----            -------------
      --------    ------      -------
37s       37s     1   {default-scheduler }                    No
rmal      Scheduled   Successfully assigned liveness-exec to wor
ker0
36s       36s     1   {kubelet worker0}   spec.containers{livene
ss}   Normal      Pulling     pulling image "gcr.io/google_conta
iners/busybox"
36s       36s     1   {kubelet worker0}   spec.containers{livene
ss}   Normal      Pulled      Successfully pulled image "gcr.io/
google_containers/busybox"
36s       36s     1   {kubelet worker0}   spec.containers{livene
ss}   Normal      Created     Created container with docker id 8
6849c15382e; Security:[seccomp=unconfined]
36s       36s     1   {kubelet worker0}   spec.containers{livene
ss}   Normal      Started     Started container with docker id 8
6849c15382e
2s        2s      1   {kubelet worker0}   spec.containers{livene
ss}   Warning     Unhealthy   Liveness probe failed: cat: can't
open '/tmp/healthy': No such file or directory
```

再等30秒，确认容器已经重启：

```
kubectl get pod liveness-exec
```

从输出结果来 `RESTARTS` 值加1了。

```
NAME              READY     STATUS    RESTARTS   AGE
liveness-exec     1/1       Running   1          1m
```

定义一个liveness HTTP请求

我们还可以使用HTTP GET请求作为liveness probe。下面是一个基于 `gcr.io/google_containers/liveness` 镜像运行了一个容器的Pod的例子 `http-liveness.yaml` ：

```yaml
apiVersion: v1
kind: Pod
metadata:
  labels:
    test: liveness
  name: liveness-http
spec:
  containers:
  - name: liveness
    args:
    - /server
    image: gcr.io/google_containers/liveness
    livenessProbe:
      httpGet:
        path: /healthz
        port: 8080
        httpHeaders:
          - name: X-Custom-Header
            value: Awesome
      initialDelaySeconds: 3
      periodSeconds: 3
```

该配置文件只定义了一个容器， `livenessProbe` 指定kubelete需要每隔3秒执行一次liveness probe。 `initialDelaySeconds` 指定kubelet在该执行第一次探测之前需要等待3秒钟。该探针将向容器中的server的8080端口发送一个HTTP GET请求。如果server的 `/healthz` 路径的handler返回一个成功的返回码，kubelet就会认定该容器是活着的并且很健康。如果返回失败的返回码，kubelet将杀掉该容器并重启它。

任何大于200小于400的返回码都会认定是成功的返回码。其他返回码都会被认为是失败的返回码。

查看该server的源码：server.go.

最开始的10秒该容器是活着的， `/healthz` handler返回200的状态码。这之后将返回500的返回码。

```
http.HandleFunc("/healthz", func(w http.ResponseWriter, r *http.
Request) {
    duration := time.Now().Sub(started)
    if duration.Seconds() > 10 {
        w.WriteHeader(500)
        w.Write([]byte(fmt.Sprintf("error: %v", duration.Seconds(
))))
    } else {
        w.WriteHeader(200)
        w.Write([]byte("ok"))
    }
})
```

容器启动3秒后，kubelet开始执行健康检查。第一次健康监测会成功，但是10秒后，健康检查将失败，kubelet将杀掉和重启容器。

创建一个Pod来测试一下HTTP liveness检测：

```
kubectl create -f https://k8s.io/docs/tasks/configure-pod-contai
ner/http-liveness.yaml
```

After 10 seconds, view Pod events to verify that liveness probes have failed and the Container has been restarted:

10秒后，查看Pod的event，确认liveness probe失败并重启了容器。

```
kubectl describe pod liveness-http
```

定义TCP liveness探针

第三种liveness probe使用TCP Socket。 使用此配置，kubelet将尝试在指定端口上打开容器的套接字。 如果可以建立连接，容器被认为是健康的，如果不能就认为是失败的。

```
apiVersion: v1
kind: Pod
metadata:
  name: goproxy
  labels:
    app: goproxy
spec:
  containers:
  - name: goproxy
    image: gcr.io/google_containers/goproxy:0.1
    ports:
    - containerPort: 8080
    readinessProbe:
      tcpSocket:
        port: 8080
      initialDelaySeconds: 5
      periodSeconds: 10
    livenessProbe:
      tcpSocket:
        port: 8080
      initialDelaySeconds: 15
      periodSeconds: 20
```

如您所见，TCP检查的配置与HTTP检查非常相似。 此示例同时使用了readiness和liveness probe。 容器启动后5秒钟，kubelet将发送第一个readiness probe。 这将尝试连接到端口8080上的goproxy容器。如果探测成功，则该pod将被标记为就绪。Kubelet将每隔10秒钟执行一次该检查。

除了readiness probe之外，该配置还包括liveness probe。 容器启动15秒后，kubelet将运行第一个liveness probe。 就像readiness probe一样，这将尝试连接到goproxy容器上的8080端口。如果liveness probe失败，容器将重新启动。

使用命名的端口

可以使用命名的ContainerPort作为HTTP或TCP liveness检查：

```
ports:
- name: liveness-port
  containerPort: 8080
  hostPort: 8080

livenessProbe:
  httpGet:
  path: /healthz
  port: liveness-port
```

定义readiness探针

有时，应用程序暂时无法对外部流量提供服务。 例如，应用程序可能需要在启动期间加载大量数据或配置文件。 在这种情况下，你不想杀死应用程序，但你也不想发送请求。 Kubernetes提供了readiness probe来检测和减轻这些情况。 Pod中的容器可以报告自己还没有准备，不能处理Kubernetes服务发送过来的流量。

Readiness probe的配置跟liveness probe很像。唯一的不同是使用 `readinessProbe` 而不是 `livenessProbe` 。

```
readinessProbe:
  exec:
    command:
    - cat
    - /tmp/healthy
  initialDelaySeconds: 5
  periodSeconds: 5
```

Readiness probe的HTTP和TCP的探测器配置跟liveness probe一样。

Readiness和livenss probe可以并行用于同一容器。 使用两者可以确保流量无法到达未准备好的容器，并且容器在失败时重新启动。

配置Probe

Probe中有很多精确和详细的配置，通过它们你能准确的控制liveness和readiness检查：

- `initialDelaySeconds` ：容器启动后第一次执行探测是需要等待多少秒。
- `periodSeconds` ：执行探测的频率。默认是10秒，最小1秒。

- `timeoutSeconds` ：探测超时时间。默认1秒，最小1秒。
- `successThreshold` ：探测失败后，最少连续探测成功多少次才被认定为成功。默认是1。对于liveness必须是1。最小值是1。
- `failureThreshold` ：探测成功后，最少连续探测失败多少次才被认定为失败。默认是3。最小值是1。

HTTP probe中可以给 `httpGet` 设置其他配置项：

- `host` ：连接的主机名，默认连接到pod的IP。你可能想在http header中设置"Host"而不是使用IP。
- `scheme` ：连接使用的schema，默认HTTP。
- `path` : 访问的HTTP server的path。
- `httpHeaders` ：自定义请求的header。HTTP运行重复的header。
- `port` ：访问的容器的端口名字或者端口号。端口号必须介于1和65525之间。

对于HTTP探测器，kubelet向指定的路径和端口发送HTTP请求以执行检查。Kubelet将probe发送到容器的IP地址，除非地址被 `httpGet` 中的可选 `host` 字段覆盖。 在大多数情况下，你不想设置主机字段。 有一种情况下你可以设置它。 假设容器在127.0.0.1上侦听，并且Pod的 `hostNetwork` 字段为true。 然后，在 `httpGet` 下的 `host` 应该设置为127.0.0.1。 如果你的pod依赖于虚拟主机，这可能是更常见的情况，你不应该是用 `host` ，而是应该在 `httpHeaders` 中设置 `Host` 头。

参考

- 关于 Container Probes 的更多信息

配置 Pod 的 Service Account

Service account 为 Pod 中的进程提供身份信息。

本文是关于 *Service Account* 的用户指南，管理指南另见 *Service Account* 的集群管理指南 。

注意：本文档描述的关于 *Serivce Account* 的行为只有当您按照 *Kubernetes* 项目建议的方式搭建起集群的情况下才有效。您的集群管理员可能在您的集群中有自定义配置，这种情况下该文档可能并不适用。

当您（真人用户）访问集群（例如使用 `kubectl` 命令）时，apiserver 会将您认证为一个特定的 User Account（目前通常是 `admin` ，除非您的系统管理员自定义了集群配置）。Pod 容器中的进程也可以与 apiserver 联系。 当它们在联系 apiserver 的时候，它们会被认证为一个特定的 Service Account（例如 `default` ）。

使用默认的 Service Account 访问 API server

当您创建 pod 的时候，如果您没有指定一个 service account，系统会自动得在与该pod 相同的 namespace 下为其指派一个 `default` service account。如果您获取刚创建的 pod 的原始 json 或 yaml 信息（例如使用 `kubectl get pods/podename -o yaml` 命令），您将看到 `spec.serviceAccountName` 字段已经被设置为 automatically set 。

您可以在 pod 中使用自动挂载的 service account 凭证来访问 API，如 Accessing the Cluster 中所描述。

Service account 是否能够取得访问 API 的许可取决于您使用的 授权插件和策略。

在 1.6 以上版本中，您可以选择取消为 service account 自动挂载 API 凭证，只需在 service account 中设置 `automountServiceAccountToken: false` ：

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: build-robot
automountServiceAccountToken: false
...
```

在 1.6 以上版本中，您也可以选择只取消单个 pod 的 API 凭证自动挂载：

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  serviceAccountName: build-robot
  automountServiceAccountToken: false
  ...
```

如果在 pod 和 service account 中同时设置了 `automountServiceAccountToken` , pod 设置中的优先级更高。

使用多个Service Account

每个 namespace 中都有一个默认的叫做 `default` 的 service account 资源。

您可以使用以下命令列出 namespace 下的所有 serviceAccount 资源。

```
$ kubectl get serviceAccounts
NAME       SECRETS    AGE
default    1          1d
```

您可以像这样创建一个 ServiceAccount 对象：

```
$ cat > /tmp/serviceaccount.yaml <<EOF
apiVersion: v1
kind: ServiceAccount
metadata:
  name: build-robot
EOF
$ kubectl create -f /tmp/serviceaccount.yaml
serviceaccount "build-robot" created
```

如果您看到如下的 service account 对象的完整输出信息：

```
$ kubectl get serviceaccounts/build-robot -o yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  creationTimestamp: 2015-06-16T00:12:59Z
  name: build-robot
  namespace: default
  resourceVersion: "272500"
  selfLink: /api/v1/namespaces/default/serviceaccounts/build-rob
ot
  uid: 721ab723-13bc-11e5-aec2-42010af0021e
secrets:
- name: build-robot-token-bvbk5
```

然后您将看到有一个 token 已经被自动创建，并被 service account 引用。

您可以使用授权插件来 设置 service account 的权限 。

设置非默认的 service account，只需要在 pod 的 `spec.serviceAccountName` 字段中将name设置为您想要用的 service account 名字即可。

在 pod 创建之初 service account 就必须已经存在，否则创建将被拒绝。

您不能更新已创建的 pod 的 service account。

您可以清理 service account，如下所示：

```
$ kubectl delete serviceaccount/build-robot
```

手动创建 service account 的 API token

假设我们已经有了一个如上文提到的名为 ”build-robot“ 的 service account，我们手动创建一个新的 secret。

```
$ cat > /tmp/build-robot-secret.yaml <<EOF
apiVersion: v1
kind: Secret
metadata:
  name: build-robot-secret
  annotations:
    kubernetes.io/service-account.name: build-robot
type: kubernetes.io/service-account-token
EOF
$ kubectl create -f /tmp/build-robot-secret.yaml
secret "build-robot-secret" created
```

现在您可以确认下新创建的 secret 取代了 "build-robot" 这个 service account 原来的 API token。

所有已不存在的 service account 的 token 将被 token controller 清理掉。

```
$ kubectl describe secrets/build-robot-secret
Name:    build-robot-secret
Namespace:   default
Labels:    <none>
Annotations:   kubernetes.io/service-account.name=build-robot,kub
ernetes.io/service-account.uid=870ef2a5-35cf-11e5-8d06-005056b45
392

Type: kubernetes.io/service-account-token

Data
====
ca.crt: 1220 bytes
token: ...
namespace: 7 bytes
```

> 注意该内容中的 `token` 被省略了。

为 service account 添加 ImagePullSecret

首先，创建一个 imagePullSecret，详见这里。

然后，确认已创建。如：

```
$ kubectl get secrets myregistrykey
NAME             TYPE                              DATA      AGE
myregistrykey    kubernetes.io/.dockerconfigjson   1         1d
```

然后，修改 namespace 中的默认 service account 使用该 secret 作为 imagePullSecret。

```
kubectl patch serviceaccount default -p '{"imagePullSecrets": [{
"name": "myregistrykey"}]}'
```

Vi 交互过程中需要手动编辑：

```
$ kubectl get serviceaccounts default -o yaml > ./sa.yaml
$ cat sa.yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  creationTimestamp: 2015-08-07T22:02:39Z
  name: default
  namespace: default
  resourceVersion: "243024"
  selfLink: /api/v1/namespaces/default/serviceaccounts/default
  uid: 052fb0f4-3d50-11e5-b066-42010af0d7b6
secrets:
- name: default-token-uudge
$ vi sa.yaml
[editor session not shown]
[delete line with key "resourceVersion"]
[add lines with "imagePullSecret:"]
$ cat sa.yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  creationTimestamp: 2015-08-07T22:02:39Z
  name: default
  namespace: default
  selfLink: /api/v1/namespaces/default/serviceaccounts/default
  uid: 052fb0f4-3d50-11e5-b066-42010af0d7b6
secrets:
- name: default-token-uudge
imagePullSecrets:
- name: myregistrykey
$ kubectl replace serviceaccount default -f ./sa.yaml
serviceaccounts/default
```

现在，所有当前 namespace 中新创建的 pod 的 spec 中都会增加如下内容：

```
spec:
  imagePullSecrets:
  - name: myregistrykey
```

Secret 配置

`Secret` 对象类型用来保存敏感信息，例如密码、OAuth 令牌和 ssh key。将这些信息放在 `secret` 中比放在 `pod` 的定义中或者 docker 镜像中来说更加安全和灵活。参阅 Secret 设计文档 获取更多详细信息。

Secret 概览

Secret 是一种包含少量敏感信息例如密码、token 或 key 的对象。这样的信息可能会被放在 Pod spec 中或者镜像中；将其放在一个 secret 对象中可以更好地控制它的用途，并降低意外暴露的风险。

用户可以创建 secret，同时系统也创建了一些 secret。

要使用 secret，pod 需要引用 secret。Pod 可以用两种方式使用 secret：作为 volume 中的文件被挂载到 pod 中的一个或者多个容器里，或者当 kubelet 为 pod 拉取镜像时使用。

内置 secret

Service Account 使用 API 凭证自动创建和附加 secret

Kubernetes 自动创建包含访问 API 凭据的 secret，并自动修改您的 pod 以使用此类型的 secret。

如果需要，可以禁用或覆盖自动创建和使用API凭据。但是，如果您需要的只是安全地访问 apiserver，我们推荐这样的工作流程。

参阅 Service Account 文档获取关于 Service Account 如何工作的更多信息。

创建您自己的 Secret

使用 kubectl 创建 Secret

假设有些 pod 需要访问数据库。这些 pod 需要使用的用户名和密码在您本地机器的 `./username.txt` 和 `./password.txt` 文件里。

```
# Create files needed for rest of example.
$ echo -n "admin" > ./username.txt
$ echo -n "1f2d1e2e67df" > ./password.txt
```

`kubectl create secret` 命令将这些文件打包到一个 Secret 中并在 API server 中创建了一个对象。

```
$ kubectl create secret generic db-user-pass --from-file=./usern
ame.txt --from-file=./password.txt
secret "db-user-pass" created
```

您可以这样检查刚创建的 secret：

```
$ kubectl get secrets
NAME                  TYPE                                  DATA
      AGE
db-user-pass          Opaque                                2
      51s

$ kubectl describe secrets/db-user-pass
Name:            db-user-pass
Namespace:       default
Labels:          <none>
Annotations:     <none>

Type:            Opaque

Data
====
password.txt:    12 bytes
username.txt:    5 bytes
```

请注意，默认情况下， `get` 和 `describe` 命令都不会显示文件的内容。这是为了防止将 secret 中的内容被意外暴露给从终端日志记录中刻意寻找它们的人。

请参阅 解码 secret 了解如何查看它们的内容。

手动创建 Secret

您也可以先以 json 或 yaml 格式在文件中创建一个 secret 对象，然后创建该对象。

每一项必须是 base64 编码：

```
$ echo -n "admin" | base64
YWRtaW4=
$ echo -n "1f2d1e2e67df" | base64
MWYyZDFlMmU2N2Rm
```

现在可以像这样写一个 secret 对象：

```
apiVersion: v1
kind: Secret
metadata:
  name: mysecret
type: Opaque
data:
  username: YWRtaW4=
  password: MWYyZDFlMmU2N2Rm
```

数据字段是一个映射。它的键必须匹配 DNS_SUBDOMAIN，前导点也是可以的。这些值可以是任意数据，使用 base64 进行编码。

使用 `kubectl create` 创建 secret：

```
$ kubectl create -f ./secret.yaml
secret "mysecret" created
```

编码注意：secret 数据的序列化 JSON 和 YAML 值使用 base64 编码成字符串。换行符在这些字符串中无效，必须省略。当在Darwin/OS X上使用 `base64` 实用程序时，用户应避免使用 `-b` 选项来拆分长行。另外，对于 Linux用户如果 `-w` 选项不可用的话，应该添加选项 `-w 0` 到 `base64` 命令或管道 `base64 | tr -d '\n'` 。

解码 Secret

可以使用 `kubectl get secret` 命令获取 secret。例如，获取在上一节中创建的 secret：

```
$ kubectl get secret mysecret -o yaml
apiVersion: v1
data:
  username: YWRtaW4=
  password: MWYyZDFlMmU2N2Rm
kind: Secret
metadata:
  creationTimestamp: 2016-01-22T18:41:56Z
  name: mysecret
  namespace: default
  resourceVersion: "164619"
  selfLink: /api/v1/namespaces/default/secrets/mysecret
  uid: cfee02d6-c137-11e5-8d73-42010af00002
type: Opaque
```

解码密码字段：

```
$ echo "MWYyZDFlMmU2N2Rm" | base64 --decode
1f2d1e2e67df
```

使用 Secret

Secret 可以作为数据卷被挂载，或作为环境变量暴露出来以供 pod 中的容器使用。它们也可以被系统的其他部分使用，而不直接暴露在 pod 内。例如，它们可以保存凭据，系统的其他部分应该用它来代表您与外部系统进行交互。

在 Pod 中使用 Secret 文件

在 Pod 中的 volume 里使用 Secret：

1. 创建一个 secret 或者使用已有的 secret。多个 pod 可以引用同一个 secret。
2. 修改您的 pod 的定义在 `spec.volumes[]` 下增加一个 volume。可以给这个 volume 随意命名，它的 `spec.volumes[].secret.secretName` 必须等于 secret 对象的名字。
3. 将 `spec.containers[].volumeMounts[]` 加到需要用到该 secret 的容器中。指定 `spec.containers[].volumeMounts[].readOnly = true` 和 `spec.containers[].volumeMounts[].mountPath` 为您想要该 secret 出现的尚未使用的目录。
4. 修改您的镜像并且／或者命令行让程序从该目录下寻找文件。Secret 的 `data` 映射中的每一个键都成为了 `mountPath` 下的一个文件名。

这是一个在 pod 中使用 volume 挂在 secret 的例子：

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  containers:
  - name: mypod
    image: redis
    volumeMounts:
    - name: foo
      mountPath: "/etc/foo"
      readOnly: true
  volumes:
  - name: foo
    secret:
      secretName: mysecret
```

您想要用的每个 secret 都需要在 `spec.volumes` 中指明。

如果 pod 中有多个容器，每个容器都需要自己的 `volumeMounts` 配置块，但是每个 secret 只需要一个 `spec.volumes`。

您可以打包多个文件到一个 secret 中，或者使用的多个 secret，怎样方便就怎样来。

向特性路径映射 secret 密钥

我们还可以控制 Secret key 映射在 volume 中的路径。您可以使用 `spec.volumes[].secret.items` 字段修改每个 key 的目标路径：

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  containers:
  - name: mypod
    image: redis
    volumeMounts:
    - name: foo
      mountPath: "/etc/foo"
      readOnly: true
  volumes:
  - name: foo
    secret:
      secretName: mysecret
      items:
      - key: username
        path: my-group/my-username
```

将会发生什么呢：

- `username` secret 存储在 `/etc/foo/my-group/my-username` 文件中而不是 `/etc/foo/username` 中。
- `password` secret 没有被影射

如果使用了 `spec.volumes[].secret.items`，只有在 `items` 中指定的 key 被影射。要使用 secret 中所有的 key，所有这些都必须列在 `items` 字段中。所有列出的密钥必须存在于相应的 secret 中。否则，不会创建卷。

Secret 文件权限

您还可以指定 secret 将拥有的权限模式位文件。如果不指定，默认使用 `0644`。您可以为整个保密卷指定默认模式，如果需要，可以覆盖每个密钥。

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  containers:
  - name: mypod
    image: redis
    volumeMounts:
    - name: foo
      mountPath: "/etc/foo"
  volumes:
  - name: foo
    secret:
      secretName: mysecret
      defaultMode: 256
```

然后，secret 将被挂载到 `/etc/foo` 目录，所有通过该 secret volume 挂载创建的文件的权限都是 `0400` 。

请注意，JSON 规范不支持八进制符号，因此使用 256 值作为 0400 权限。如果您使用 yaml 而不是 json 作为 pod，则可以使用八进制符号以更自然的方式指定权限。

您还可以是用映射，如上一个示例，并为不同的文件指定不同的权限，如下所示：

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  containers:
  - name: mypod
    image: redis
    volumeMounts:
    - name: foo
      mountPath: "/etc/foo"
  volumes:
  - name: foo
    secret:
      secretName: mysecret
      items:
      - key: username
        path: my-group/my-username
        mode: 511
```

在这种情况下，导致 `/etc/foo/my-group/my-username` 的文件的权限值为 `0777` 。由于 JSON 限制，必须以十进制格式指定模式。

请注意，如果稍后阅读此权限值可能会以十进制格式显示。

从 **Volume** 中消费 **secret** 值

在挂载的 secret volume 的容器内，secret key 将作为文件，并且 secret 的值使用 base-64 解码并存储在这些文件中。这是在上面的示例容器内执行的命令的结果：

```
$ ls /etc/foo/
username
password
$ cat /etc/foo/username
admin
$ cat /etc/foo/password
1f2d1e2e67df
```

容器中的程序负责从文件中读取 secret。

挂载的 **secret** 被自动更新

当已经在 volume 中消被消费的 secret 被更新时，被映射的 key 也将被更新。

Kubelet 在周期性同步时检查被挂载的 secret 是不是最新的。但是，它正在使用其基于本地 ttl 的缓存来获取当前的 secret 值。结果是，当 secret 被更新的时刻到将新的 secret 映射到 pod 的时刻的总延迟可以与 kubelet 中的secret 缓存的 kubelet sync period + ttl 一样长。

Secret 作为环境变量

将 secret 作为 pod 中的环境变量使用：

1. 创建一个 secret 或者使用一个已存在的 secret。多个 pod 可以引用同一个 secret。
2. 在每个容器中修改您想要使用 secret key 的 Pod 定义，为要使用的每个 secret key 添加一个环境变量。消费secret key 的环境变量应填充 secret 的名称，并键入 `env[x].valueFrom.secretKeyRef` 。
3. 修改镜像并／或者命令行，以便程序在指定的环境变量中查找值。

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: secret-env-pod
spec:
  containers:
  - name: mycontainer
    image: redis
    env:
      - name: SECRET_USERNAME
        valueFrom:
          secretKeyRef:
            name: mysecret
            key: username
      - name: SECRET_PASSWORD
        valueFrom:
          secretKeyRef:
            name: mysecret
            key: password
  restartPolicy: Never
```

消费环境变量里的 **Secret** 值

在一个消耗环境变量 secret 的容器中，secret key 作为包含 secret 数据的 base-64 解码值的常规环境变量。这是从上面的示例在容器内执行的命令的结果：

```
$ echo $SECRET_USERNAME
admin
$ echo $SECRET_PASSWORD
1f2d1e2e67df
```

使用 imagePullSecret

imagePullSecret 是将包含 Docker（或其他）镜像注册表密码的 secret 传递给 Kubelet 的一种方式，因此可以代表您的 pod 拉取私有镜像。

手动指定 imagePullSecret

imagePullSecret 的使用在 镜像文档 中说明。

安排 imagePullSecrets 自动附加

您可以手动创建 imagePullSecret，并从 serviceAccount 引用它。使用该 serviceAccount 创建的任何 pod 和默认使用该 serviceAccount 的 pod 将会将其的 imagePullSecret 字段设置为服务帐户的 imagePullSecret 字段。有关该过程的详细说明，请参阅 将 ImagePullSecrets 添加到服务帐户 。

自动挂载手动创建的 Secret

手动创建的 secret（例如包含用于访问 github 帐户的令牌）可以根据其服务帐户自动附加到 pod。请参阅 使用 PodPreset 向 Pod 中注入信息 以获取该进程的详细说明。

详细

限制

验证 secret volume 来源确保指定的对象引用实际上指向一个类型为 Secret 的对象。因此，需要在依赖于它的任何 pod 之前创建一个 secret。

Secret API 对象驻留在命名空间中。它们只能由同一命名空间中的 pod 引用。

每个 secret 的大小限制为1MB。这是为了防止创建非常大的 secret 会耗尽 apiserver 和 kubelet 的内存。然而，创建许多较小的 secret 也可能耗尽内存。更全面得限制 secret 对内存使用的更全面的限制是计划中的功能。

Kubelet 仅支持从 API server 获取的 Pod 使用 secret。这包括使用 kubectl 创建的任何 pod，或间接通过 replication controller 创建的 pod。它不包括通过 kubelet `--manifest-url` 标志，其 `--config` 标志或其 REST API 创建的pod（这些不是创建 pod 的常用方法）。

必须先创建 secret，除非将它们标记为可选项，否则必须在将其作为环境变量在 pod 中使用之前创建 secret。对不存在的 secret 的引用将阻止其启动。

通过 `secretKeyRef` 对不存在于命名的 key 中的 key 进行引用将阻止该启动。

用于通过 `envFrom` 填充环境变量的 secret，这些环境变量具有被认为是无效环境变量名称的 key 将跳过这些键。该 pod 将被允许启动。将会有一个事件，其原因是 `InvalidVariableNames` ，该消息将包含被跳过的无效键的列表。该示例显示一个 pod，它指的是包含2个无效键，1badkey 和 2alsobad 的默认/mysecret ConfigMap。

```
$ kubectl get events
LASTSEEN   FIRSTSEEN   COUNT     NAME            KIND      SUBOB
JECT                          TYPE      REASON
0s         0s          1         dapi-test-pod   Pod
                              Warning   InvalidEnvironmentVariabl
eNames   kubelet, 127.0.0.1      Keys [1badkey, 2alsobad] from t
he EnvFrom secret default/mysecret were skipped since they are c
onsidered invalid environment variable names.
```

Secret 与 Pod 生命周期的联系

通过 API 创建的 Pod 时，不会检查应用的 secret 是否存在。一旦 Pod 被调度，kubelet 就会尝试获取该 secret 的值。如果获取不到该 secret，或者暂时无法与 API server 建立连接，kubelet 将会定期重试。Kubelet 将会报告关于 pod 的事件，并解释它无法启动的原因。一旦获取的 secret，kubelet将创建并装载一个包含它的卷。在安装所有pod的卷之前，都不会启动 pod 的容器。

使用案例

使用案例：包含 ssh 密钥的 pod

创建一个包含 ssh key 的 secret：

```
$ kubectl create secret generic ssh-key-secret --from-file=ssh-p
rivatekey=/path/to/.ssh/id_rsa --from-file=ssh-publickey=/path/t
o/.ssh/id_rsa.pub
```

安全性注意事项：发送自己的 ssh 密钥之前要仔细思考：集群的其他用户可能有权访问该密钥。使用您想要共享 Kubernetes 群集的所有用户可以访问的服务帐户，如果它们遭到入侵，可以撤销。

现在我们可以创建一个使用 ssh 密钥引用 secret 的pod，并在一个卷中使用它：

```yaml
kind: Pod
apiVersion: v1
metadata:
  name: secret-test-pod
  labels:
    name: secret-test
spec:
  volumes:
  - name: secret-volume
    secret:
      secretName: ssh-key-secret
  containers:
  - name: ssh-test-container
    image: mySshImage
    volumeMounts:
    - name: secret-volume
      readOnly: true
      mountPath: "/etc/secret-volume"
```

当容器中的命令运行时，密钥的片段将可在以下目录：

```
/etc/secret-volume/ssh-publickey
/etc/secret-volume/ssh-privatekey
```

然后容器可以自由使用密钥数据建立一个 ssh 连接。

使用案例：包含 prod/test 凭据的 pod

下面的例子说明一个 pod 消费一个包含 prod 凭据的 secret，另一个 pod 使用测试环境凭据消费 secret。

创建 secret：

```
$ kubectl create secret generic prod-db-secret --from-literal=us
ername=produser --from-literal=password=Y4nys7f11
secret "prod-db-secret" created
$ kubectl create secret generic test-db-secret --from-literal=us
ername=testuser --from-literal=password=iluvtests
secret "test-db-secret" created
```

创建 pod ：

```yaml
apiVersion: v1
kind: List
items:
- kind: Pod
  apiVersion: v1
  metadata:
    name: prod-db-client-pod
    labels:
      name: prod-db-client
  spec:
    volumes:
    - name: secret-volume
      secret:
        secretName: prod-db-secret
    containers:
    - name: db-client-container
      image: myClientImage
      volumeMounts:
      - name: secret-volume
        readOnly: true
        mountPath: "/etc/secret-volume"
- kind: Pod
  apiVersion: v1
  metadata:
    name: test-db-client-pod
    labels:
      name: test-db-client
  spec:
    volumes:
    - name: secret-volume
      secret:
        secretName: test-db-secret
    containers:
    - name: db-client-container
      image: myClientImage
      volumeMounts:
      - name: secret-volume
        readOnly: true
        mountPath: "/etc/secret-volume"
```

这两个容器将在其文件系统上显示以下文件，其中包含每个容器环境的值：

```
/etc/secret-volume/username
/etc/secret-volume/password
```

请注意，两个 pod 的 spec 配置中仅有一个字段有所不同；这有助于使用普通的 pod 配置模板创建具有不同功能的 pod。您可以使用两个 service account 进一步简化基本 pod spec：一个名为 `prod-user` 拥有 `prod-db-secret` ，另一个称为

`test-user` 拥有 `test-db-secret` 。然后，pod spec 可以缩短为，例如：

```
kind: Pod
apiVersion: v1
metadata:
  name: prod-db-client-pod
  labels:
    name: prod-db-client
spec:
  serviceAccount: prod-db-client
  containers:
  - name: db-client-container
    image: myClientImage
```

使用案例：secret 卷中以点号开头的文件

为了将数据“隐藏”起来（即文件名以点号开头的文件），简单地说让该键以一个点开始。例如，当如下 secret 被挂载到卷中：

```
kind: Secret
apiVersion: v1
metadata:
  name: dotfile-secret
data:
  .secret-file: dmFsdWUtMg0KDQo=
---
kind: Pod
apiVersion: v1
metadata:
  name: secret-dotfiles-pod
spec:
  volumes:
  - name: secret-volume
    secret:
      secretName: dotfile-secret
  containers:
  - name: dotfile-test-container
    image: gcr.io/google_containers/busybox
    command:
    - ls
    - "-l"
    - "/etc/secret-volume"
    volumeMounts:
    - name: secret-volume
      readOnly: true
      mountPath: "/etc/secret-volume"
```

`Secret-volume` 将包含一个单独的文件，叫做 `.secret-file` ，`dotfile-test-container` 的 `/etc/secret-volume/.secret-file` 路径下将有该文件。

注意

以点号开头的文件在 `ls -l` 的输出中被隐藏起来了；列出目录内容时，必须使用 `ls -la` 才能查看它们。

使用案例：Secret 仅对 pod 中的一个容器可见

考虑以下一个需要处理 HTTP 请求的程序，执行一些复杂的业务逻辑，然后使用 HMAC 签署一些消息。因为它具有复杂的应用程序逻辑，所以在服务器中可能会出现一个未被注意的远程文件读取漏洞，这可能会将私钥暴露给攻击者。

这可以在两个容器中分为两个进程：前端容器，用于处理用户交互和业务逻辑，但无法看到私钥；以及可以看到私钥的签名者容器，并且响应来自前端的简单签名请求（例如通过本地主机网络）。

使用这种分割方法，攻击者现在必须欺骗应用程序服务器才能进行任意的操作，这可能比使其读取文件更难。

最佳实践

客户端使用 secret API

当部署与 secret API 交互的应用程序时，应使用诸如 RBAC 之类的 授权策略 来限制访问。

Secret 的重要性通常不尽相同，其中许多可能只对 Kubernetes 集群内（例如 service account 令牌）和对外部系统造成影响。即使一个应用程序可以理解其期望的与之交互的 secret 的权力，但是同一命名空间中的其他应用程序也可以使这些假设无效。

由于这些原因，在命名空间中 `watch` 和 `list` secret 的请求是非常强大的功能，应该避免这样的行为，因为列出 secret 可以让客户端检查所有 secret 是否在该命名空间中。在群集中 `watch` 和 `list` 所有 secret 的能力应该只保留给最有特权的系统级组件。

需要访问 secrets API 的应用程序应该根据他们需要的 secret 执行 `get` 请求。这允许管理员限制对所有 secret 的访问，同时设置 白名单访问 应用程序需要的各个实例。

为了提高循环获取的性能，客户端可以设计引用 secret 的资源，然后 `watch` 资源，在引用更改时重新请求 secret。此外，还提出了一种 ”批量监控“ API 来让客户端 `watch` 每个资源，该功能可能会在将来的 Kubernetes 版本中提供。

安全属性

保护

因为 `secret` 对象可以独立于使用它们的 `pod` 而创建，所以在创建、查看和编辑 pod 的流程中 secret 被暴露的风险较小。系统还可以对 `secret` 对象采取额外的预防措施，例如避免将其写入到磁盘中可能的位置。

只有当节点上的 pod 需要用到该 secret 时，该 secret 才会被发送到该节点上。它不会被写入磁盘，而是存储在 tmpfs 中。一旦依赖于它的 pod 被删除，它就被删除。

在大多数 Kubernetes 项目维护的发行版中，用户与 API server 之间的通信以及从 API server 到 kubelet 的通信都受到 SSL/TLS 的保护。通过这些通道传输时，secret 受到保护。

节点上的 secret 数据存储在 tmpfs 卷中，因此不会传到节点上的其他磁盘。

同一节点上的很多个 pod 可能拥有多个 secret。但是，只有 pod 请求的 secret 在其容器中才是可见的。因此，一个 pod 不能访问另一个 Pod 的 secret。

Pod 中有多个容器。但是，pod 中的每个容器必须请求其挂载卷中的 secret 卷才能在容器内可见。这可以用于 在 Pod 级别构建安全分区。

风险

- API server 的 secret 数据以纯文本的方式存储在 etcd 中；因此：
    - 管理员应该限制 admin 用户访问 etcd；
    - API server 中的 secret 数据位于 etcd 使用的磁盘上；管理员可能希望在不再使用时擦除/粉碎 etcd 使用的磁盘

- 如果您将 secret 数据编码为 base64 的清单（JSON 或 YAML）文件，共享该文件或将其检入代码库，这样的话该密码将会被泄露。 Base64 编码不是一种加密方式，一样也是纯文本。
- 应用程序在从卷中读取 secret 后仍然需要保护 secret 的值，例如不会意外记录或发送给不信任方。
- 可以创建和使用 secret 的 pod 的用户也可以看到该 secret 的值。即使 API server 策略不允许用户读取 secret 对象，用户也可以运行暴露 secret 的 pod。
- 如果运行了多个副本，那么这些 secret 将在它们之间共享。默认情况下，etcd 不能保证与 SSL/TLS 的对等通信，尽管可以进行配置。
- 目前，任何节点的 root 用户都可以通过模拟 kubelet 来读取 API server 中的任何 secret。只有向实际需要它们的节点发送 secret 才能限制单个节点的根漏洞的影响，该功能还在计划中。

原文地址：https://github.com/rootsongjc/kubernetes.github.io/blob/master/docs/concepts/configuration/secret.md

管理namespace中的资源配额

当用多个团队或者用户共用同一个集群的时候难免会有资源竞争的情况发生，这时候就需要对不同团队或用户的资源使用配额做出限制。

开启资源配额限制功能

目前有两种资源分配管理相关的控制策略插件 `ResourceQuota` 和 `LimitRange`。

要启用它们只要 API Server 的启动配置的 `KUBE_ADMISSION_CONTROL` 参数中加入了 `ResourceQuota` 的设置，这样就给集群开启了资源配额限制功能，加入 `LimitRange` 可以用来限制一个资源申请的范围限制，参考 为 namesapce 配置默认的内存请求与限额 和 在 namespace 中配置默认的CPU请求与限额。

两种控制策略的作用范围都是对于某一 namespace，`ResourceQuota` 用来限制 namespace 中所有的 Pod 占用的总的资源 request 和 limit，而 `LimitRange` 是用来设置 namespace 中 Pod 的默认的资源 request 和 limit 值。

资源配额分为三种类型：

- 计算资源配额
- 存储资源配额
- 对象数量配额

关于资源配额的详细信息请参考 kubernetes 官方文档 资源配额。

示例

我们为 `spark-cluster` 这个 namespace 设置 `ResouceQuota` 和 `LimitRange`。

以下 yaml 文件可以在 kubernetes-handbook 的 `manifests/spark-with-kubernetes-native-scheduler` 目录下找到。

配置计算资源配额

配置文件：`spark-compute-resources.yaml`

```yaml
apiVersion: v1
kind: ResourceQuota
metadata:
  name: compute-resources
  namespace: spark-cluster
spec:
  hard:
    pods: "20"
    requests.cpu: "20"
    requests.memory: 100Gi
    limits.cpu: "40"
    limits.memory: 200Gi
```

要想查看该配置只要执行：

```
kubectl -n spark-cluster describe resourcequota compute-resource
s
```

配置对象数量限制

配置文件：`spark-object-counts.yaml`

```yaml
apiVersion: v1
kind: ResourceQuota
metadata:
  name: object-counts
  namespace: spark-cluster
spec:
  hard:
    configmaps: "10"
    persistentvolumeclaims: "4"
    replicationcontrollers: "20"
    secrets: "10"
    services: "10"
    services.loadbalancers: "2"
```

配置CPU和内存LimitRange

配置文件：`spark-limit-range.yaml`

```yaml
apiVersion: v1
kind: LimitRange
metadata:
  name: mem-limit-range
spec:
  limits:
  - default:
      memory: 50Gi
      cpu: 5
    defaultRequest:
      memory: 1Gi
      cpu: 1
    type: Container
```

- `default` 即 limit 的值
- `defaultRequest` 即 request 的值

参考

- 资源配额
- 为命名空间配置默认的内存请求与限额
- 在命名空间中配置默认的CPU请求与限额

命令使用

Kubernetes 中的 kubectl 及其他管理命令使用。

docker用户过度到kubectl命令行指南

对于没有使用过 kubernetes 的 docker 用户，如何快速掌握 kubectl 命令？

在本文中，我们将向 docker-cli 用户介绍 Kubernetes 命令行如何与 api 进行交互。该命令行工具——kubectl，被设计成 docker-cli 用户所熟悉的样子，但是它们之间又存在一些必要的差异。该文档将向您展示每个 docker 子命令和 kubectl 与其等效的命令。

在使用 kubernetes 集群的时候，docker 命令通常情况是不需要用到的，只有在调试程序或者容器的时候用到，我们基本上使用 kubectl 命令即可，所以在操作 kubernetes 的时候我们抛弃原先使用 docker 时的一些观念。

docker run

如何运行一个 nginx Deployment 并将其暴露出来？ 查看 kubectl run 。

使用 docker 命令：

```
$ docker run -d --restart=always -e DOMAIN=cluster --name nginx-
app -p 80:80 nginx
a9ec34d9878748d2f33dc20cb25c714ff21da8d40558b45bfaec9955859075d0
$ docker ps
CONTAINER ID        IMAGE               COMMAND                  C
REATED              STATUS              PORTS
     NAMES
a9ec34d98787        nginx               "nginx -g 'daemon of   2
 seconds ago        Up 2 seconds        0.0.0.0:80->80/tcp, 443/t
cp   nginx-app
```

使用 kubectl 命令：

```
# start the pod running nginx
$ kubectl run --image=nginx nginx-app --port=80 --env="DOMAIN=cl
uster"
deployment "nginx-app" created
```

在大于等于 1.2 版本 Kubernetes 集群中，使用 `kubectl run` 命令将创建一个名为 "nginx-app" 的 Deployment。如果您运行的是老版本，将会创建一个 replication controller。 如果您想沿用旧的行为，使用 `--generation=run/v1` 参数，这样就

会创建 replication controller。查看 `kubectl run` 获取更多详细信息。

```
# expose a port through with a service
$ kubectl expose deployment nginx-app --port=80 --name=nginx-htt
p
service "nginx-http" exposed
```

在 kubectl 命令中，我们创建了一个 Deployment，这将保证有 N 个运行 nginx 的 pod（N 代表 spec 中声明的 replica 数，默认为 1）。我们还创建了一个 service，使用 selector 匹配具有相应的 selector 的 Deployment。查看 快速开始 获取更多信息。

默认情况下镜像会在后台运行，与 `docker run -d ...` 类似，如果您想在前台运行，使用：

```
kubectl run [-i] [--tty] --attach <name> --image=<image>
```

与 `docker run ...` 不同的是，如果指定了 `--attach` ，我们将连接到 `stdin` ， `stdout` 和 `stderr` ，而不能控制具体连接到哪个输出流（ `docker -a ...` ）。

因为我们使用 Deployment 启动了容器，如果您终止了连接到的进程（例如 `ctrl-c` ），容器将会重启，这跟 `docker run -it` 不同。 如果想销毁该 Deployment（和它的 pod），您需要运行 `kubeclt delete deployment <name>` 。

docker ps

如何列出哪些正在运行？查看 kubectl get。

使用 docker 命令：

```
$ docker ps
CONTAINER ID        IMAGE               COMMAND                C
REATED              STATUS              PORTS
     NAMES
a9ec34d98787        nginx               "nginx -g 'daemon of   A
bout an hour ago    Up About an hour    0.0.0.0:80->80/tcp, 443/t
cp   nginx-app
```

使用 kubectl 命令：

```
$ kubectl get po
NAME                 READY     STATUS    RESTARTS   AGE
nginx-app-5jyvm      1/1       Running   0          1h
```

docker attach

如何连接到已经运行在容器中的进程？查看 kubectl attach。

使用 docker 命令：

```
$ docker ps
CONTAINER ID        IMAGE                COMMAND                C
REATED              STATUS               PORTS
     NAMES
a9ec34d98787        nginx                "nginx -g 'daemon of   8
 minutes ago        Up 8 minutes         0.0.0.0:80->80/tcp, 443/t
cp   nginx-app
$ docker attach a9ec34d98787
...
```

使用 kubectl 命令：

```
$ kubectl get pods
NAME                 READY     STATUS    RESTARTS   AGE
nginx-app-5jyvm      1/1       Running   0          10m
$ kubectl attach -it nginx-app-5jyvm
...
```

docker exec

如何在容器中执行命令？查看 kubectl exec。

使用 docker 命令：

```
$ docker ps
CONTAINER ID        IMAGE               COMMAND                  C
REATED              STATUS              PORTS
     NAMES
a9ec34d98787        nginx               "nginx -g 'daemon of   8
 minutes ago        Up 8 minutes        0.0.0.0:80->80/tcp, 443/t
cp   nginx-app
$ docker exec a9ec34d98787 cat /etc/hostname
a9ec34d98787
```

使用 kubectl 命令：

```
$ kubectl get po
NAME              READY     STATUS    RESTARTS   AGE
nginx-app-5jyvm   1/1       Running   0          10m
$ kubectl exec nginx-app-5jyvm -- cat /etc/hostname
nginx-app-5jyvm
```

执行交互式命令怎么办？

使用 docker 命令：

```
$ docker exec -ti a9ec34d98787 /bin/sh
# exit
```

使用 kubectl 命令：

```
$ kubectl exec -ti nginx-app-5jyvm -- /bin/sh
# exit
```

更多信息请查看 获取运行中容器的 Shell 环境。

docker logs

如何查看运行中进程的 stdout/stderr？查看 kubectl logs。

使用 docker 命令：

```
$ docker logs -f a9e
192.168.9.1 - - [14/Jul/2015:01:04:02 +0000] "GET / HTTP/1.1" 20
0 612 "-" "curl/7.35.0" "-"
192.168.9.1 - - [14/Jul/2015:01:04:03 +0000] "GET / HTTP/1.1" 20
0 612 "-" "curl/7.35.0" "-"
```

使用 kubectl 命令：

```
$ kubectl logs -f nginx-app-zibvs
10.240.63.110 - - [14/Jul/2015:01:09:01 +0000] "GET / HTTP/1.1"
200 612 "-" "curl/7.26.0" "-"
10.240.63.110 - - [14/Jul/2015:01:09:02 +0000] "GET / HTTP/1.1"
200 612 "-" "curl/7.26.0" "-"
```

现在是时候提一下 pod 和容器之间的细微差别了；默认情况下如果 pod 中的进程退出 pod 也不会终止，相反它将会重启该进程。这类似于 docker run 时的 `--restart=always` 选项， 这是主要差别。在 docker 中，进程的每个调用的输出都是被连接起来的，但是对于 kubernetes，每个调用都是分开的。要查看以前在 kubernetes 中执行的输出，请执行以下操作：

```
$ kubectl logs --previous nginx-app-zibvs
10.240.63.110 - - [14/Jul/2015:01:09:01 +0000] "GET / HTTP/1.1"
200 612 "-" "curl/7.26.0" "-"
10.240.63.110 - - [14/Jul/2015:01:09:02 +0000] "GET / HTTP/1.1"
200 612 "-" "curl/7.26.0" "-"
```

查看 记录和监控集群活动 获取更多信息。

docker stop 和 docker rm

如何停止和删除运行中的进程？查看 kubectl delete。

使用 docker 命令：

```
$ docker ps
CONTAINER ID        IMAGE               COMMAND                  C
REATED             STATUS              PORTS
     NAMES
a9ec34d98787        nginx               "nginx -g 'daemon of   2
2 hours ago         Up 22 hours         0.0.0.0:80->80/tcp, 443/t
cp   nginx-app
$ docker stop a9ec34d98787
a9ec34d98787
$ docker rm a9ec34d98787
a9ec34d98787
```

使用 kubectl 命令：

```
$ kubectl get deployment nginx-app
NAME        DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
nginx-app   1         1         1            1           2m
$ kubectl get po -l run=nginx-app
NAME                         READY     STATUS    RESTARTS   AGE
nginx-app-2883164633-aklf7   1/1       Running   0          2m
$ kubectl delete deployment nginx-app
deployment "nginx-app" deleted
$ kubectl get po -l run=nginx-app
# Return nothing
```

请注意，我们不直接删除 pod。使用 kubectl 命令，我们要删除拥有该 pod 的 Deployment。如果我们直接删除pod，Deployment 将会重新创建该 pod。

docker login

在 kubectl 中没有对 `docker login` 的直接模拟。如果您有兴趣在私有镜像仓库中使用 Kubernetes，请参阅 使用私有镜像仓库。

docker version

如何查看客户端和服务端的版本？查看 kubectl version。

使用 docker 命令：

```
$ docker version
Client version: 1.7.0
Client API version: 1.19
Go version (client): go1.4.2
Git commit (client): 0baf609
OS/Arch (client): linux/amd64
Server version: 1.7.0
Server API version: 1.19
Go version (server): go1.4.2
Git commit (server): 0baf609
OS/Arch (server): linux/amd64
```

使用 kubectl 命令：

```
$ kubectl version
Client Version: version.Info{Major:"1", Minor:"6", GitVersion:"v1.6.9+a3d1dfa6f4335", GitCommit:"9b77fed11a9843ce3780f70dd251e92901c43072", GitTreeState:"dirty", BuildDate:"2017-08-29T20:32:58Z", OpenPaasKubernetesVersion:"v1.03.02", GoVersion:"go1.7.5", Compiler:"gc", Platform:"linux/amd64"}
Server Version: version.Info{Major:"1", Minor:"6", GitVersion:"v1.6.9+a3d1dfa6f4335", GitCommit:"9b77fed11a9843ce3780f70dd251e92901c43072", GitTreeState:"dirty", BuildDate:"2017-08-29T20:32:58Z", OpenPaasKubernetesVersion:"v1.03.02", GoVersion:"go1.7.5", Compiler:"gc", Platform:"linux/amd64"}
```

docker info

如何获取有关环境和配置的各种信息？查看 kubectl cluster-info。

使用 docker 命令：

```
$ docker info
Containers: 40
Images: 168
Storage Driver: aufs
 Root Dir: /usr/local/google/docker/aufs
 Backing Filesystem: extfs
 Dirs: 248
 Dirperm1 Supported: false
Execution Driver: native-0.2
Logging Driver: json-file
Kernel Version: 3.13.0-53-generic
Operating System: Ubuntu 14.04.2 LTS
CPUs: 12
Total Memory: 31.32 GiB
Name: k8s-is-fun.mtv.corp.google.com
ID: ADUV:GCYR:B3VJ:HMPO:LNPQ:KD5S:YKFQ:76VN:IANZ:7TFV:ZBF4:BYJO
WARNING: No swap limit support
```

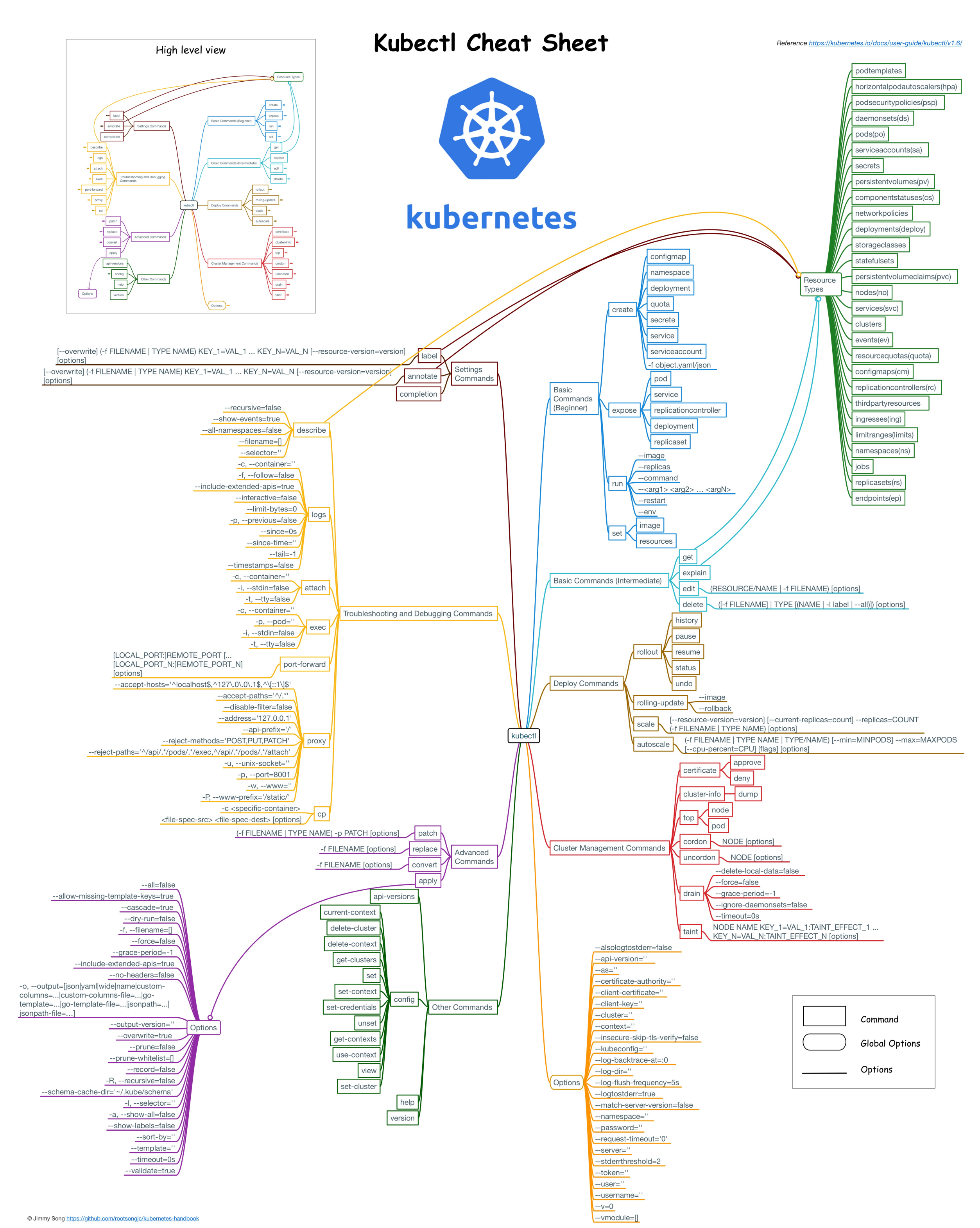
使用 kubectl 命令：

```
$ kubectl cluster-info
Kubernetes master is running at https://108.59.85.141
KubeDNS is running at https://108.59.85.141/api/v1/namespaces/ku
be-system/services/kube-dns/proxy
KubeUI is running at https://108.59.85.141/api/v1/namespaces/kub
e-system/services/kube-ui/proxy
Grafana is running at https://108.59.85.141/api/v1/namespaces/ku
be-system/services/monitoring-grafana/proxy
Heapster is running at https://108.59.85.141/api/v1/namespaces/k
ube-system/services/monitoring-heapster/proxy
InfluxDB is running at https://108.59.85.141/api/v1/namespaces/k
ube-system/services/monitoring-influxdb/proxy
```

原文地址：https://github.com/rootsongjc/kubernetes.github.io/blob/master/docs/user-guide/docker-cli-to-kubectl.md

Kubectl命令概览

Kubernetes提供的kubectl命令是与集群交互最直接的方式，v1.6版本的kubectl命令参考图如下：



图片 - *kubectl cheatsheet*

Kubectl的子命令主要分为8个类别：

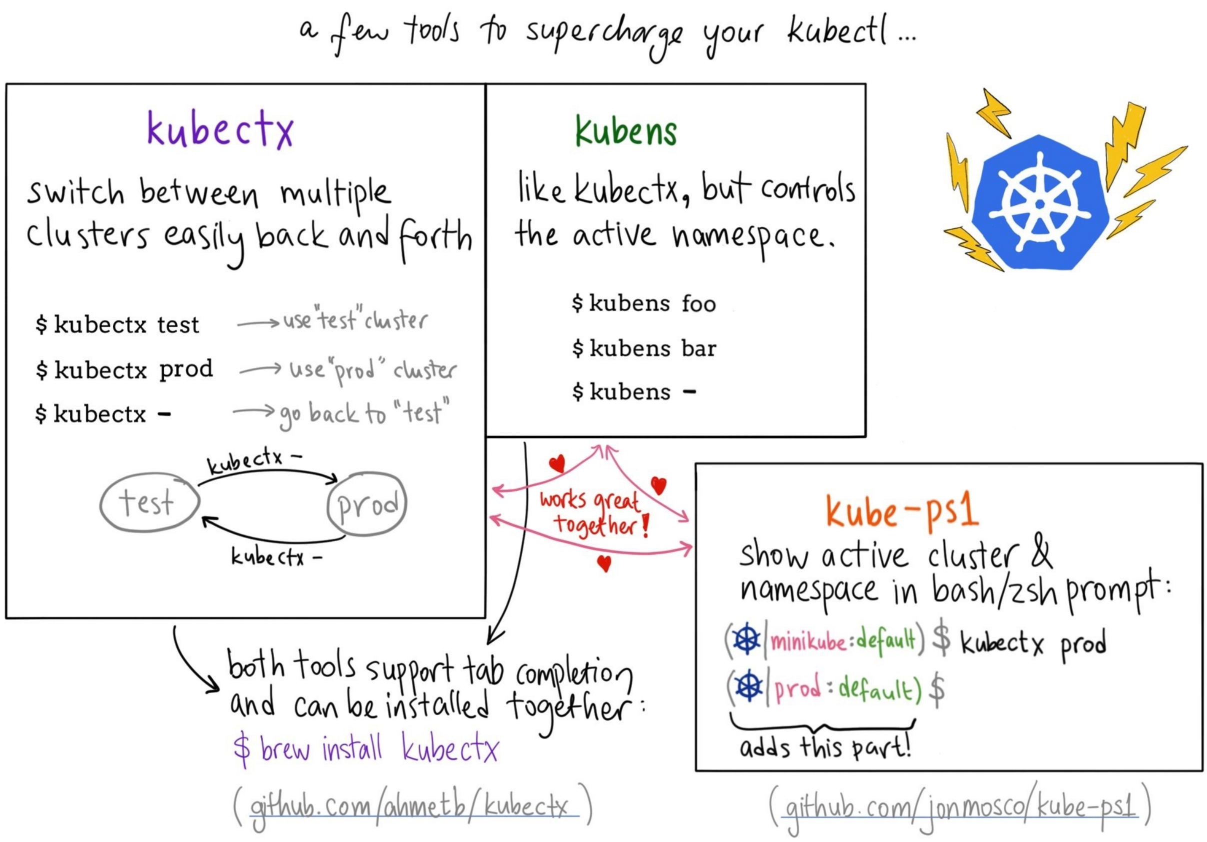
- 基础命令（初学者都会使用的）

- 基础命令（中级）
- 部署命令
- 集群管理命令
- 故障排查和调试命令
- 高级命令
- 设置命令
- 其他命令

熟悉这些命令有助于大家来操作和管理kubernetes集群。

命令行提示

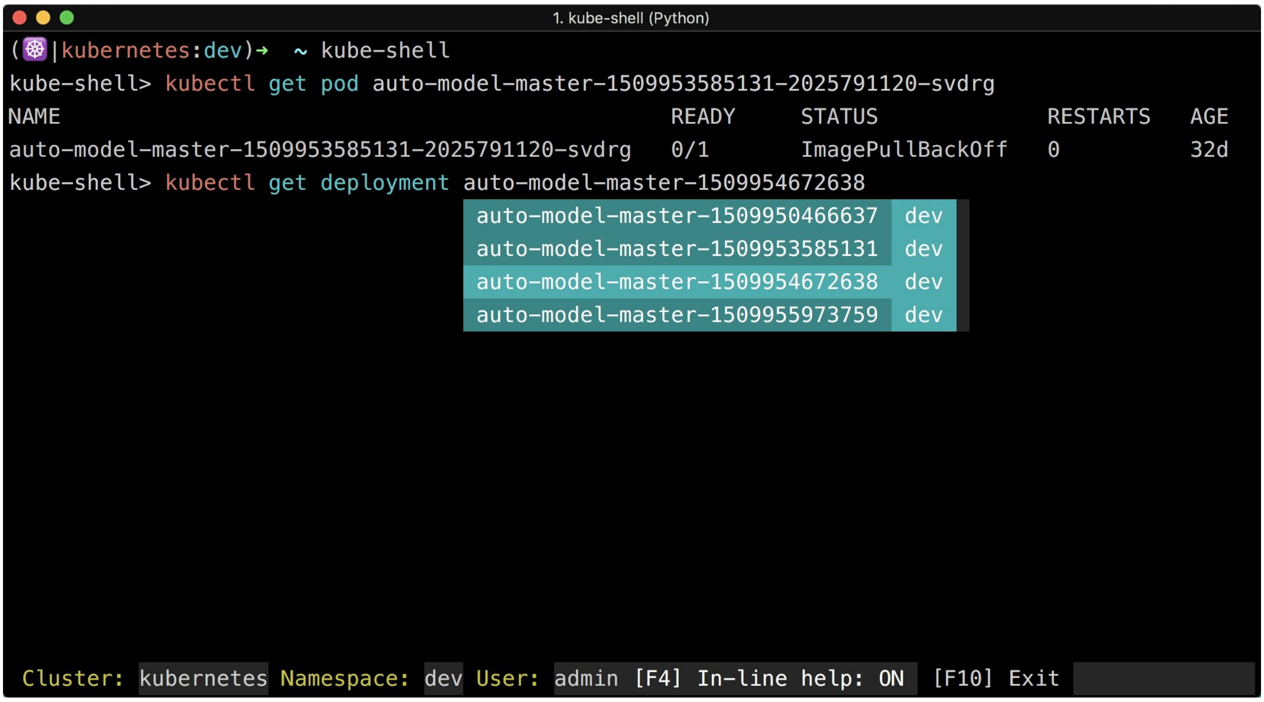
为了使用kubectl命令更加高效，我们可以选择安装一下开源软件来增加操作kubectl命令的快捷方式，同时为kubectl命令增加命令提示。



图片 - 增加*kubeclt*命令的工具（图片来自网络）

- kubectx：用于切换kubernetes context
- kube-ps1：为命令行终端增加 `$PROMPT` 字段
- kube-shell：交互式带命令提示的kubectl终端

全部配置完成后的kubectl终端如下图所示：



图片 - 增强的*kubectl*命令

kube-shell

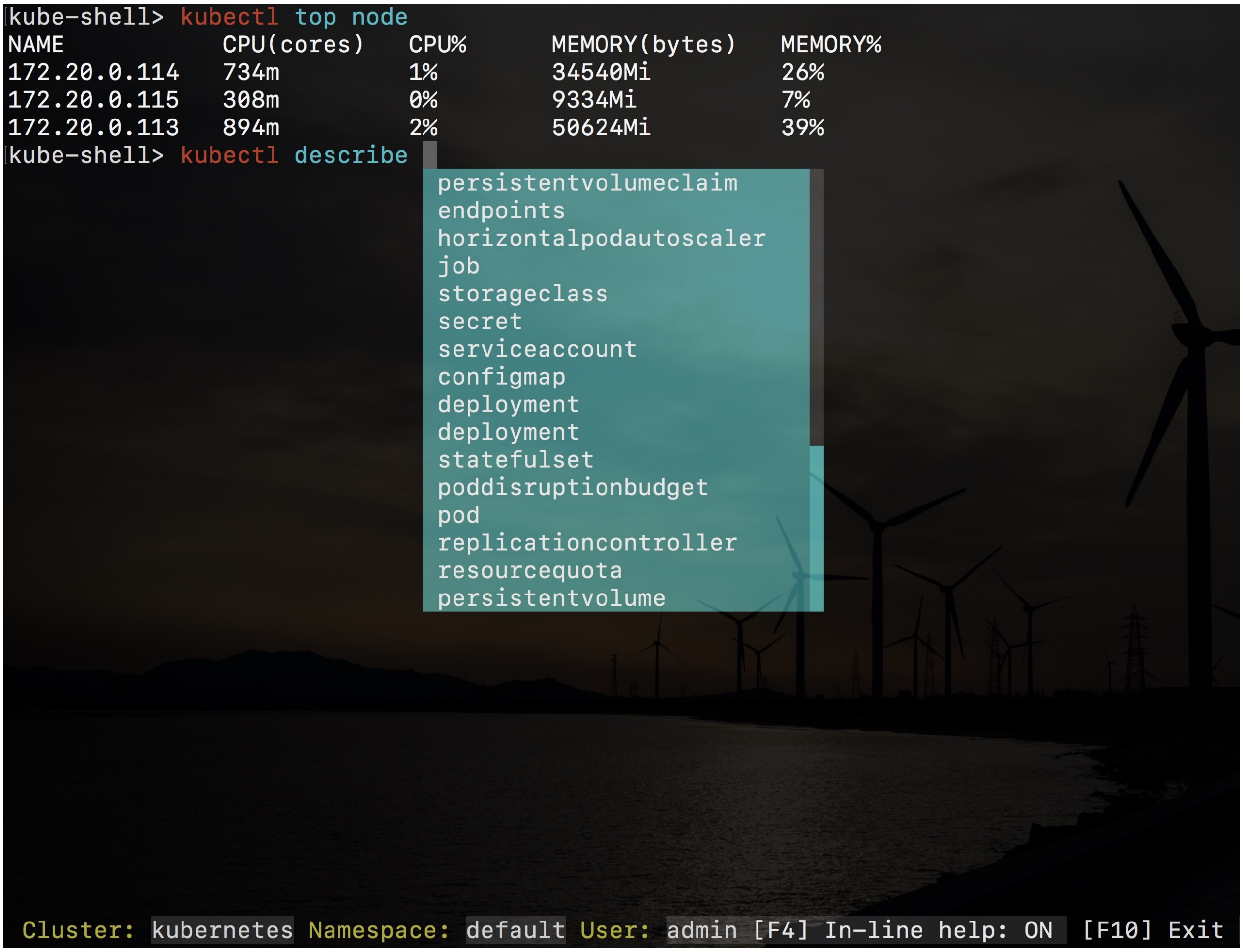
开源项目kube-shell可以为kubectl提供自动的命令提示和补全，使用起来特别方便，推荐给大家。

Kube-shell有以下特性：

- 命令提示，给出命令的使用说明
- 自动补全，列出可选命令并可以通过tab键自动补全，支持模糊搜索
- 高亮
- 使用tab键可以列出可选的对象
- vim模式

Mac下安装

```
pip install kube-shell --user -U
```



图片 - *kube-shell*页面

kubectl的身份认证

Kubernetes中存在三种安全认证方式：

- **CA**证书：API server与其它几个组件之间都是通过这种方式认证的
- **HTTP base**：即在API server的启动参数中指定的 `--token-auth-file=/etc/kubernetes/token.csv` 文件中明文的用户、组、密码和UID配置
- **bearer token**：HTTP请求中 `header` 中传递的 `Autorization:Bearer token` ，这个token通常保存在创建角色跟 `serviceaccount` 绑定的时候生成的secret中。

kubectl通过读取 `kubeconfig` 文件中的配置信息在向API server发送请求的时候同时传递认证信息，同时支持CA证书和bearer token的认证方式，请参考使用kubeconfig文件配置跨集群认证。

终端下kubectl命令自动补全

建议使用oh-my-zsh，增加对kubectl命令自动补全支持。

修改 `~/.zshrc` 文件，增加如下两行：

```
plugins=(kubectl)
source <(kubectl completion zsh)
```

保存后重启终端即可生效。

参考：Install and Set Up kubectl

kubectl 命令技巧大全

Kubctl 命令是操作 kubernetes 集群的最直接和最 skillful 的途径，这个60多MB大小的二进制文件，到底有啥能耐呢？请看下文：

Kubectl 自动补全

```
$ source <(kubectl completion bash) # setup autocomplete in bash
, bash-completion package should be installed first.
$ source <(kubectl completion zsh)  # setup autocomplete in zsh
```

Kubectl 上下文和配置

设置 `kubectl` 命令交互的 kubernetes 集群并修改配置信息。参阅 使用 kubeconfig 文件进行跨集群验证 获取关于配置文件的详细信息。

```
$ kubectl config view # 显示合并后的 kubeconfig 配置

# 同时使用多个 kubeconfig 文件并查看合并后的配置
$ KUBECONFIG=~/.kube/config:~/.kube/kubconfig2 kubectl config vi
ew

# 获取 e2e 用户的密码
$ kubectl config view -o jsonpath='{.users[?(@.name == "e2e")].u
ser.password}'

$ kubectl config current-context              # 显示当前的上下文
$ kubectl config use-context my-cluster-name  # 设置默认上下文为 my
-cluster-name

# 向 kubeconf 中增加支持基本认证的新集群
$ kubectl config set-credentials kubeuser/foo.kubernetes.com --u
sername=kubeuser --password=kubepassword

# 使用指定的用户名和 namespace 设置上下文
$ kubectl config set-context gce --user=cluster-admin --namespace
=foo \
  && kubectl config use-context gce
```

创建对象

Kubernetes 的清单文件可以使用 json 或 yaml 格式定义。可以以 `.yaml` 、 `.yml` 、或者 `.json` 为扩展名。

```
$ kubectl create -f ./my-manifest.yaml           # 创建资源
$ kubectl create -f ./my1.yaml -f ./my2.yaml     # 使用多个文件创建
资源
$ kubectl create -f ./dir                        # 使用目录下的所有
清单文件来创建资源
$ kubectl create -f https://git.io/vPieo         # 使用 url 来创建
资源
$ kubectl run nginx --image=nginx                # 启动一个 nginx
 实例
$ kubectl explain pods,svc                       # 获取 pod 和 sv
c 的文档

# 从 stdin 输入中创建多个 YAML 对象
$ cat <<EOF | kubectl create -f -
apiVersion: v1
kind: Pod
metadata:
  name: busybox-sleep
spec:
  containers:
  - name: busybox
    image: busybox
    args:
    - sleep
    - "1000000"
---
apiVersion: v1
kind: Pod
metadata:
  name: busybox-sleep-less
spec:
  containers:
  - name: busybox
    image: busybox
    args:
    - sleep
    - "1000"
EOF

# 创建包含几个 key 的 Secret
$ cat <<EOF | kubectl create -f -
apiVersion: v1
kind: Secret
metadata:
  name: mysecret
type: Opaque
data:
  password: $(echo "s33msi4" | base64)
  username: $(echo "jane" | base64)
EOF
```

显示和查找资源

```
# Get commands with basic output
$ kubectl get services                          # 列出所有 namesp
ace 中的所有 service
$ kubectl get pods --all-namespaces             # 列出所有 namesp
ace 中的所有 pod
$ kubectl get pods -o wide                      # 列出所有 pod 并
显示详细信息
$ kubectl get deployment my-dep                 # 列出指定 deploy
ment
$ kubectl get pods --include-uninitialized      # 列出该 namespac
e 中的所有 pod 包括未初始化的

# 使用详细输出来描述命令
$ kubectl describe nodes my-node
$ kubectl describe pods my-pod

$ kubectl get services --sort-by=.metadata.name # List Services
Sorted by Name

# 根据重启次数排序列出 pod
$ kubectl get pods --sort-by='.status.containerStatuses[0].resta
rtCount'

# 获取所有具有 app=cassandra 的 pod 中的 version 标签
$ kubectl get pods --selector=app=cassandra rc -o \
  jsonpath='{.items[*].metadata.labels.version}'

# 获取所有节点的 ExternalIP
$ kubectl get nodes -o jsonpath='{.items[*].status.addresses[?(@
.type=="ExternalIP")].address}'

# 列出属于某个 PC 的 Pod 的名字
# “jq”命令用于转换复杂的 jsonpath，参考 https://stedolan.github.io/j
q/
$ sel=${$(kubectl get rc my-rc --output=json | jq -j '.spec.sele
ctor | to_entries | .[] | "\(.key)=\(.value),"')%?}
$ echo $(kubectl get pods --selector=$sel --output=jsonpath={.it
ems..metadata.name})

# 查看哪些节点已就绪
$ JSONPATH='{range .items[*]}{@.metadata.name}:{range @.status.c
onditions[*]}{@.type}={@.status};{end}{end}' \
 && kubectl get nodes -o jsonpath="$JSONPATH" | grep "Ready=True"


# 列出当前 Pod 中使用的 Secret
$ kubectl get pods -o json | jq '.items[].spec.containers[].env[
]?.valueFrom.secretKeyRef.name' | grep -v null | sort | uniq
```

更新资源

```
$ kubectl rolling-update frontend-v1 -f frontend-v2.json
   # 滚动更新 pod frontend-v1
$ kubectl rolling-update frontend-v1 frontend-v2 --image=image:v
2  # 更新资源名称并更新镜像
$ kubectl rolling-update frontend --image=image:v2
   # 更新 frontend pod 中的镜像
$ kubectl rolling-update frontend-v1 frontend-v2 --rollback
   # 退出已存在的进行中的滚动更新
$ cat pod.json | kubectl replace -f -
   # 基于 stdin 输入的 JSON 替换 pod

# 强制替换，删除后重新创建资源。会导致服务中断。
$ kubectl replace --force -f ./pod.json

# 为 nginx RC 创建服务，启用本地 80 端口连接到容器上的 8000 端口
$ kubectl expose rc nginx --port=80 --target-port=8000

# 更新单容器 pod 的镜像版本（tag）到 v4
$ kubectl get pod mypod -o yaml | sed 's/\(image: myimage\):.*$/
\1:v4/' | kubectl replace -f -

$ kubectl label pods my-pod new-label=awesome
   # 添加标签
$ kubectl annotate pods my-pod icon-url=http://goo.gl/XXBTWq
   # 添加注解
$ kubectl autoscale deployment foo --min=2 --max=10
   # 自动扩展 deployment “foo”
```

修补资源

使用策略合并补丁并修补资源。

```
$ kubectl patch node k8s-node-1 -p '{"spec":{"unschedulable":true}}' # 部分更新节点

# 更新容器镜像； spec.containers[*].name 是必须的，因为这是合并的关键字
$ kubectl patch pod valid-pod -p '{"spec":{"containers":[{"name":"kubernetes-serve-hostname","image":"new image"}]}}'

# 使用具有位置数组的 json 补丁更新容器镜像
$ kubectl patch pod valid-pod --type='json' -p='[{"op": "replace", "path": "/spec/containers/0/image", "value":"new image"}]'

# 使用具有位置数组的 json 补丁禁用 deployment 的 livenessProbe
$ kubectl patch deployment valid-deployment  --type json   -p='[{"op": "remove", "path": "/spec/template/spec/containers/0/livenessProbe"}]'
```

编辑资源

在编辑器中编辑任何 API 资源。

```
$ kubectl edit svc/docker-registry                      # 编辑名为 docker-registry 的 service
$ KUBE_EDITOR="nano" kubectl edit svc/docker-registry   # 使用其它编辑器
```

Scale 资源

```
$ kubectl scale --replicas=3 rs/foo
    # Scale a replicaset named 'foo' to 3
$ kubectl scale --replicas=3 -f foo.yaml
    # Scale a resource specified in "foo.yaml" to 3
$ kubectl scale --current-replicas=2 --replicas=3 deployment/mysql  # If the deployment named mysql's current size is 2, scale mysql to 3
$ kubectl scale --replicas=5 rc/foo rc/bar rc/baz
    # Scale multiple replication controllers
```

删除资源

```
$ kubectl delete -f ./pod.json
            # 删除 pod.json 文件中定义的类型和名称的 pod
$ kubectl delete pod,service baz foo
            # 删除名为“baz”的 pod 和名为“foo”的 service
$ kubectl delete pods,services -l name=myLabel
            # 删除具有 name=myLabel 标签的 pod 和 serivce
$ kubectl delete pods,services -l name=myLabel --include-uniniti
alized      # 删除具有 name=myLabel 标签的 pod 和 service，包括尚未初
始化的
$ kubectl -n my-ns delete po,svc --all
            # 删除 my-ns namespace 下的所有 pod 和 serivce，包括尚未
初始化的
```

与运行中的 Pod 交互

```
$ kubectl logs my-pod                                    # dump 输出
 pod 的日志（stdout）
$ kubectl logs my-pod -c my-container                    # dump 输出
 pod 中容器的日志（stdout，pod 中有多个容器的情况下使用）
$ kubectl logs -f my-pod                                 # 流式输出
pod 的日志（stdout）
$ kubectl logs -f my-pod -c my-container                 # 流式输出
pod 中容器的日志（stdout，pod 中有多个容器的情况下使用）
$ kubectl run -i --tty busybox --image=busybox -- sh     # 交互式 sh
ell 的方式运行 pod
$ kubectl attach my-pod -i                               # 连接到运行
中的容器
$ kubectl port-forward my-pod 5000:6000                  # 转发 pod
中的 6000 端口到本地的 5000 端口
$ kubectl exec my-pod -- ls /                            # 在已存在的
容器中执行命令（只有一个容器的情况下）
$ kubectl exec my-pod -c my-container -- ls /            # 在已存在的
容器中执行命令（pod 中有多个容器的情况下）
$ kubectl top pod POD_NAME --containers                  # 显示指定
pod 和容器的指标度量
```

与节点和集群交互

```
$ kubectl cordon my-node                                                # 标记 my-node 不可调度
$ kubectl drain my-node                                                 # 清空 my-node 以待维护
$ kubectl uncordon my-node                                              # 标记 my-node 可调度
$ kubectl top node my-node                                              # 显示 my-node 的指标度量
$ kubectl cluster-info                                                  # 显示 master 和服务的地址
$ kubectl cluster-info dump                                             # 将当前集群状态输出到 stdout

$ kubectl cluster-info dump --output-directory=/path/to/cluster-state   # 将当前集群状态输出到 /path/to/cluster-state

# 如果该键和影响的污点（taint）已存在，则使用指定的值替换
$ kubectl taint nodes foo dedicated=special-user:NoSchedule
```

资源类型

下表列出的是 kubernetes 中所有支持的类型和缩写的别名。

资源类型	缩写别名
`clusters`	
`componentstatuses`	`cs`
`configmaps`	`cm`
`daemonsets`	`ds`
`deployments`	`deploy`
`endpoints`	`ep`
`event`	`ev`
`horizontalpodautoscalers`	`hpa`
`ingresses`	`ing`
`jobs`	
`limitranges`	`limits`
`namespaces`	`ns`
`networkpolicies`	
`nodes`	`no`
`statefulsets`	
`persistentvolumeclaims`	`pvc`
`persistentvolumes`	`pv`
`pods`	`po`
`podsecuritypolicies`	`psp`
`podtemplates`	
`replicasets`	`rs`
`replicationcontrollers`	`rc`
`resourcequotas`	`quota`
`cronjob`	
`secrets`	
`serviceaccount`	`sa`
`services`	`svc`
`storageclasses`	
`thirdpartyresources`	

格式化输出

要以特定的格式向终端窗口输出详细信息，可以在 `kubectl` 命令中添加 `-o` 或者 `-output` 标志。

输出格式	描述
`-o=custom-columns=<spec>`	使用逗号分隔的自定义列列表打印表格
`-o=custom-columns-file=<filename>`	使用 文件中的自定义列模板打印表格
`-o=json`	输出 JSON 格式的 API 对象
`-o=jsonpath=<template>`	打印 jsonpath 表达式中定义的字段
`-o=jsonpath-file=<filename>`	打印由 文件中的 jsonpath 表达式定义的字段
`-o=name`	仅打印资源名称
`-o=wide`	以纯文本格式输出任何附加信息，对于 Pod ，包含节点名称
`-o=yaml`	输出 YAML 格式的 API 对象

Kubectl 详细输出和调试

使用 `-v` 或 `--v` 标志跟着一个整数来指定日志级别。这里 描述了通用的 kubernetes 日志约定和相关的日志级别。

详细等级	描述
`--v=0`	总是对操作人员可见。
`--v=1`	合理的默认日志级别，如果您不需要详细输出。
`--v=2`	可能与系统的重大变化相关的，有关稳定状态的信息和重要的日志信息。这是对大多数系统推荐的日志级别。
`--v=3`	有关更改的扩展信息。
`--v=4`	调试级别详细输出。
`--v=6`	显示请求的资源。
`--v=7`	显示HTTP请求的header。
`--v=8`	显示HTTP请求的内容。

参考

- Kubectl 概览
- JsonPath 手册

本文是对官方文档的中文翻译，原文地址：https://kubernetes.io/docs/user-guide/kubectl-cheatsheet/

使用etcdctl访问kubernetes数据

Kubenretes1.6中使用etcd V3版本的API，使用 `etcdctl` 直接 `ls` 的话只能看到 `/kube-centos` 一个路径。需要在命令前加上 `ETCDCTL_API=3` 这个环境变量才能看到kuberentes在etcd中保存的数据。

```
ETCDCTL_API=3 etcdctl get /registry/namespaces/default -w=json|p
ython -m json.tool
```

- `-w` 指定输出格式

将得到这样的json的结果：


```
{
    "count": 1,
    "header": {
        "cluster_id": 12091028579527406772,
        "member_id": 16557816780141026208,
        "raft_term": 36,
        "revision": 29253467
    },
    "kvs": [
        {
            "create_revision": 5,
            "key": "L3JlZ2lzdHJ5L25hbWVzcGFjZXMvZGVmYXVsdA==",
            "mod_revision": 5,
            "value": "azhzAAoPCgJ2MRIJTmFtZXNwYWNlEmIKSAoHZGVmYX
VsdBIAGgAiACokZTU2YzMzMDgtMWVhOC0xMWU3LThjZDctZjRlOWQ0OWY4ZWQwMg
A4AEILCIn4sscFEKOg9xd6ABIMCgprdWJlcm5ldGVzGggKBkFjdGl2ZRoAIgA=",
            "version": 1
        }
    ]
}
```


使用 `--prefix` 可以看到所有的子目录，如查看集群中的namespace：

```
ETCDCTL_API=3 etcdctl get /registry/namespaces --prefix -w=json|
python -m json.tool
```

输出结果中可以看到所有的namespace。

```
{
    "count": 8,
    "header": {
        "cluster_id": 12091028579527406772,
        "member_id": 16557816780141026208,
        "raft_term": 36,
        "revision": 29253722
    },
    "kvs": [
        {
            "create_revision": 24310883,
            "key": "L3JlZ2lzdHJ5L25hbWVzcGFjZXMvYXV0b21vZGVs",
            "mod_revision": 24310883,
            "value": "azhzAAoPCgJ2MRIJTmFtZXNwYWNlEmQKSgoJYXV0b2
1vZGVsEgAaACIAKiQ1MjczOTU1ZC1iMzEyLTExZTctOTcwYy1mNGU5ZDQ5ZjhlZD
AyADgAQgsI7fSWzwUQ6Jv1Z3oAEgwKCmt1YmVybmV0ZXMaCAoGQWN0aXZlGgAiAA
==",
            "version": 1
        },
        {
            "create_revision": 21387676,
            "key": "L3JlZ2lzdHJ5L25hbWVzcGFjZXMvYnJhbmQ=",
            "mod_revision": 21387676,
            "value": "azhzAAoPCgJ2MRIJTmFtZXNwYWNlEmEKRwoFYnJhbm
QSABoAIgAqJGNkZmQ1Y2NmLWExYzktMTFlNy05NzBjLWY0ZTlkNDlmOGVkMDIAOA
BCDAjR9qLOBRDYn83XAXoAEgwKCmt1YmVybmV0ZXMaCAoGQWN0aXZlGgAiAA==",
            "version": 1
        },
        {
            "create_revision": 5,
            "key": "L3JlZ2lzdHJ5L25hbWVzcGFjZXMvZGVmYXVsdA==",
            "mod_revision": 5,
            "value": "azhzAAoPCgJ2MRIJTmFtZXNwYWNlEmIKSAoHZGVmYX
VsdBIAGgAiACokZTU2YzMzMDgtMWVhOC0xMWU3LThjZDctZjRlOWQ0OWY4ZWQwMg
A4AEILCIn4sscFEKOg9xd6ABIMCgprdWJlcm5ldGVzGggKBkFjdGl2ZRoAIgA=",
            "version": 1
        },
        {
            "create_revision": 18504694,
            "key": "L3JlZ2lzdHJ5L25hbWVzcGFjZXMvZGV2",
            "mod_revision": 24310213,
            "value": "azhzAAoPCgJ2MRIJTmFtZXNwYWNlEmwKUgoDZGV2Eg
AaACIAKiQyOGRlMGVjNS04ZTEzLTExZTctOTcwYy1mNGU5ZDQ5ZjhlZDAyADgAQg
wI89CezQUQ0v2fuQNaCwoEbmFtZRIDZGV2egASDAoKa3ViZXJuZXRlcxoICgZBY3
RpdmUaACIA",
            "version": 4
        },
        {
            "create_revision": 10,
            "key": "L3JlZ2lzdHJ5L25hbWVzcGFjZXMva3ViZS1wdWJsaWM="
,
            "mod_revision": 10,
```

```
            "value": "azhzAAoPCgJ2MRIJTmFtZXNwYWNlEmcKTQoLa3ViZS
1wdWJsaWMSABoAIgAqJGU1ZjhkY2I1LTFlYTgtMTFlNy04Y2Q3LWY0ZTlkNDlmOG
VkMDIAOABCDAiJ+LLHBRDdrsDPA3oAEgwKCmt1YmVybmV0ZXMaCAoGQWN0aXZlGg
AiAA==",
            "version": 1
        },
        {
            "create_revision": 2,
            "key": "L3JlZ2lzdHJ5L25hbWVzcGFjZXMva3ViZS1zeXN0ZW0="
,
            "mod_revision": 2,
            "value": "azhzAAoPCgJ2MRIJTmFtZXNwYWNlEmYKTAoLa3ViZS
1zeXN0ZW0SABoAIgAqJGU1NmFhMDVkLTFlYTgtMTFlNy04Y2Q3LWY0ZTlkNDlmOG
VkMDIAOABCCwiJ+LLHBRDoq9ASegASDAoKa3ViZXJuZXRlcxoICgZBY3RpdmUaAC
IA",
            "version": 1
        },
        {
            "create_revision": 3774247,
            "key": "L3JlZ2lzdHJ5L25hbWVzcGFjZXMvc3BhcmstY2x1c3Rl
cg==",
            "mod_revision": 3774247,
            "value": "azhzAAoPCgJ2MRIJTmFtZXNwYWNlEoABCmYKDXNwYX
JrLWNsdXN0ZXISABoAIgAqJDMyNjY3ZDVjLTM0YWMtMTFlNy1iZmJkLThhZjFlM2
E3YzViZDIAOABCDAiA1cbIBRDU3YuAAVoVCgRuYW1lEg1zcGFyay1jbHVzdGVyeg
ASDAoKa3ViZXJuZXRlcxoICgZBY3RpdmUaACIA",
            "version": 1
        },
        {
            "create_revision": 15212191,
            "key": "L3JlZ2lzdHJ5L25hbWVzcGFjZXMveWFybi1jbHVzdGVy"
,
            "mod_revision": 15212191,
            "value": "azhzAAoPCgJ2MRIJTmFtZXNwYWNlEn0KYwoMeWFybi
1jbHVzdGVyEgAaACIAKiQ2YWNhNjk1Yi03N2Y5LTExZTctYmZiZC04YWYxZTNhN2
M1YmQyADgAQgsI1qiKzAUQkoqxDloUCgRuYW1lEgx5YXJuLWNsdXN0ZXJ6ABIMCg
prdWJlcm5ldGVzGggKBkFjdGl2ZRoAIgA=",
            "version": 1
        }
    ]
}
```

key的值是经过base64编码，需要解码后才能看到实际值，如：

```
$ echo L3JlZ2lzdHJ5L25hbWVzcGFjZXMvYXV0b21vZGVs|base64 -d
/registry/namespaces/automodel
```

etcd中kubernetes的元数据

我们使用kubectl命令获取的kubernetes的对象状态实际上是保存在etcd中的，使用下面的脚本可以获取etcd中的所有kubernetes对象的key：

> 注意，我们使用了ETCD v3版本的客户端命令来访问etcd。

```bash
#!/bin/bash
# Get kubernetes keys from etcd
export ETCDCTL_API=3
keys=`etcdctl get /registry --prefix -w json|python -m json.tool|
grep key|cut -d ":" -f2|tr -d '"'|tr -d ","`
for x in $keys;do
  echo $x|base64 -d|sort
done
```

通过输出的结果我们可以看到kubernetes的原数据是按何种结构包括在kuberentes中的，输出结果如下所示：

```
/registry/ThirdPartyResourceData/istio.io/istioconfigs/default/r
oute-rule-details-default
/registry/ThirdPartyResourceData/istio.io/istioconfigs/default/r
oute-rule-productpage-default
/registry/ThirdPartyResourceData/istio.io/istioconfigs/default/r
oute-rule-ratings-default
...
/registry/configmaps/default/namerctl-script
/registry/configmaps/default/namerd-config
/registry/configmaps/default/nginx-config
...
/registry/deployments/default/sdmk-page-sdmk
/registry/deployments/default/sdmk-payment-web
/registry/deployments/default/sdmk-report
...
```

我们可以看到所有的Kuberentes的所有元数据都保存在 `/registry` 目录下，下一层就是API对象类型（复数形式），再下一层是 `namespace` ，最后一层是对象的名字。

以下是etcd中存储的kubernetes所有的元数据类型：

```
ThirdPartyResourceData
apiextensions.k8s.io
apiregistration.k8s.io
certificatesigningrequests
clusterrolebindings
clusterroles
configmaps
controllerrevisions
controllers
daemonsets
deployments
events
horizontalpodautoscalers
ingress
limitranges
minions
monitoring.coreos.com
namespaces
persistentvolumeclaims
persistentvolumes
poddisruptionbudgets
pods
ranges
replicasets
resourcequotas
rolebindings
roles
secrets
serviceaccounts
services
statefulsets
storageclasses
thirdpartyresources
```

参考

- etcd中文文档
- etcd官方文档

集群安全性管理

Kuberentes 支持多租户，这就需要对集群的安全性进行管理。

管理集群中的TLS

在本书的最佳实践部分，我们在CentOS上部署了kuberentes集群，其中最开始又重要的一步就是创建TLS认证的，查看创建TLS证书和秘钥。很多人在进行到这一步时都会遇到各种各样千奇百怪的问题，这一步是创建集群的基础，我们有必要详细了解一下其背后的流程和原理。

概览

每个Kubernetes集群都有一个集群根证书颁发机构（CA）。 集群中的组件通常使用CA来验证API server的证书，由API服务器验证kubelet客户端证书等。为了支持这一点，CA证书包被分发到集群中的每个节点，并作为一个sercret附加分发到默认service account上。 或者，你的workload可以使用此CA建立信任。 你的应用程序可以使用类似于ACME草案的协议，使用 `certificates.k8s.io` API请求证书签名。

集群中的TLS信任

让Pod中运行的应用程序信任集群根CA通常需要一些额外的应用程序配置。 您将需要将CA证书包添加到TLS客户端或服务器信任的CA证书列表中。 例如，您可以使用golang TLS配置通过解析证书链并将解析的证书添加到 `tls.Config` 结构中的 `Certificates` 字段中，CA证书捆绑包将使用默认服务账户自动加载到pod中，路径为 `/var/run/secrets/kubernetes.io/serviceaccount/ca.crt` 。 如果您没有使用默认服务账户，请请求集群管理员构建包含您有权访问使用的证书包的configmap。

请求认证

以下部分演示如何为通过DNS访问的Kubernetes服务创建TLS证书。

步骤0. 下载安装SSL

下载cfssl工具：https://pkg.cfssl.org/.

步骤1. 创建证书签名请求

通过运行以下命令生成私钥和证书签名请求（或CSR）：

```
$ cat <<EOF | cfssl genkey - | cfssljson -bare server
{
  "hosts": [
    "my-svc.my-namespace.svc.cluster.local",
    "my-pod.my-namespace.pod.cluster.local",
    "172.168.0.24",
    "10.0.34.2"
  ],
  "CN": "my-pod.my-namespace.pod.cluster.local",
  "key": {
    "algo": "ecdsa",
    "size": 256
  }
}
EOF
```

`172.168.0.24` 是 service 的 cluster IP，`my-svc.my-namespace.svc.cluster.local` 是 service 的 DNS 名称，`10.0.34.2` 是 Pod 的 IP，`my-pod.my-namespace.pod.cluster.local` 是pod 的 DNS 名称，你可以看到以下输出：

```
2017/03/21 06:48:17 [INFO] generate received request
2017/03/21 06:48:17 [INFO] received CSR
2017/03/21 06:48:17 [INFO] generating key: ecdsa-256
2017/03/21 06:48:17 [INFO] encoded CSR
```

此命令生成两个文件; 它生成包含PEM编码的pkcs #10认证请求的 `server.csr`，以及包含仍然要创建的证书的PEM编码密钥的 `server-key.pem`。

步骤2. 创建证书签名请求对象以发送到Kubernetes API

使用以下命令创建CSR yaml文件，并发送到API server：

```
$ cat <<EOF | kubectl create -f -
apiVersion: certificates.k8s.io/v1beta1
kind: CertificateSigningRequest
metadata:
  name: my-svc.my-namespace
spec:
  groups:
  - system:authenticated
  request: $(cat server.csr | base64 | tr -d '\n')
  usages:
  - digital signature
  - key encipherment
  - server auth
EOF
```

请注意，在步骤1中创建的 `server.csr` 文件是base64编码并存储在 `.spec.request` 字段中。 我们还要求提供“数字签名”，“密钥加密”和“服务器身份验证”密钥用途的证书。 我们这里支持列出的所有关键用途和扩展的关键用途，以便您可以使用相同的API请求客户端证书和其他证书。

在API server中可以看到这些CSR处于pending状态。执行下面的命令你将可以看到：

```
$ kubectl describe csr my-svc.my-namespace
Name:                   my-svc.my-namespace
Labels:                 <none>
Annotations:            <none>
CreationTimestamp:      Tue, 21 Mar 2017 07:03:51 -0700
Requesting User:        yourname@example.com
Status:                 Pending
Subject:
        Common Name:    my-svc.my-namespace.svc.cluster.local
        Serial Number:
Subject Alternative Names:
        DNS Names:      my-svc.my-namespace.svc.cluster.local
        IP Addresses:   172.168.0.24
                        10.0.34.2
Events: <none>
```

步骤3. 获取证书签名请求

批准证书签名请求是通过自动批准过程完成的，或由集群管理员一次完成。 有关这方面涉及的更多信息，请参见下文。

步骤4. 下载签名并使用

一旦CSR被签署并获得批准，您应该看到以下内容：

```
$ kubectl get csr
NAME                   AGE       REQUESTOR                 CONDITIO
N
my-svc.my-namespace    10m       yourname@example.com      Approved
,Issued
```

你可以通过运行以下命令下载颁发的证书并将其保存到 `server.crt` 文件中：

```
$ kubectl get csr my-svc.my-namespace -o jsonpath='{.status.cert
ificate}' \
    | base64 -d > server.crt
```

现在你可以用 `server.crt` 和 `server-key.pem` 来做为keypair来启动HTTPS server。

批准证书签名请求

Kubernetes 管理员（具有适当权限）可以使用 `kubectl certificate approve` 和 `kubectl certificate deny` 命令手动批准（或拒绝）证书签名请求。但是，如果您打算大量使用此 API，则可以考虑编写自动化的证书控制器。

如果上述机器或人类使用 kubectl，批准者的作用是验证 CSR 满足如下两个要求：

1. CSR 的主体控制用于签署 CSR 的私钥。这解决了伪装成授权主体的第三方的威胁。在上述示例中，此步骤将验证该 pod 控制了用于生成 CSR 的私钥。
2. CSR 的主体被授权在请求的上下文中执行。这解决了我们加入群集的我们不期望的主体的威胁。在上述示例中，此步骤将是验证该 pod 是否被允许加入到所请求的服务中。

当且仅当满足这两个要求时，审批者应该批准 CSR，否则拒绝 CSR。

给集群管理员的一个建议

本教程假设将signer设置为服务证书API。Kubernetes controller manager提供了一个signer的默认实现。 要启用它，请将 `--cluster-signing-cert-file` 和 `--cluster-signing-key-file` 参数传递给controller manager，并配置具有证书颁发机构的密钥对的路径。

Kublet的认证授权

概览

Kubelet 的 HTTPS 端点对外暴露了用于访问不同敏感程度数据的 API，并允许您在节点或者容器内执行不同权限级别的操作。

本文档向您描述如何通过认证授权来访问 kubelet 的 HTTPS 端点。

Kubelet 认证

默认情况下，所有未被配置的其他身份验证方法拒绝的，对 kubelet 的 HTTPS 端点的请求将被视为匿名请求，并被授予 `system:anonymous` 用户名和 `system:unauthenticated` 组。

如果要禁用匿名访问并发送 `401 Unauthorized` 的未经身份验证的请求的响应：

- 启动 kubelet 时指定 `--anonymous-auth=false` 标志

如果要对 kubelet 的 HTTPS 端点启用 X509 客户端证书身份验证：

- 启动 kubelet 时指定 `--client-ca-file` 标志，提供 CA bundle 以验证客户端证书
- 启动 apiserver 时指定 `--kubelet-client-certificate` 和 `--kubelet-client-key` 标志
- 参阅 apiserver 认证文档 获取更多详细信息。

启用 API bearer token（包括 service account token）用于向 kubelet 的 HTTPS 端点进行身份验证：

- 确保在 API server 中开启了 `authentication.k8s.io/v1beta1` API 组。
- 启动 kubelet 时指定 `--authentication-token-webhook`，`--kubeconfig` 和 `--require-kubeconfig` 标志
- Kubelet 在配置的 API server 上调用 `TokenReview` API 以确定来自 bearer token 的用户信息

Kubelet 授权

接着对任何成功验证的请求（包括匿名请求）授权。默认授权模式为 `AlwaysAllow` ，允许所有请求。

细分访问 kubelet API 有很多原因：

- 启用匿名认证，但匿名用户调用 kubelet API 的能力应受到限制
- 启动 bearer token 认证，但是 API 用户（如 service account）调用 kubelet API 的能力应受到限制
- 客户端证书身份验证已启用，但只有那些配置了 CA 签名的客户端证书的用户才可以使用 kubelet API

如果要细分访问 kubelet API，将授权委托给 API server：

- 确保 API server 中启用了 `authorization.k8s.io/v1beta1` API 组
- 启动 kubelet 时指定 `--authorization-mode=Webhook` 、 `--kubeconfig` 和 `--require-kubeconfig` 标志
- kubelet 在配置的 API server 上调用 `SubjectAccessReview` API，以确定每个请求是否被授权

kubelet 使用与 apiserver 相同的 请求属性 方法来授权 API 请求。

Verb（动词）是根据传入的请求的 HTTP 动词确定的：

HTTP 动词	request 动词
POST	create
GET, HEAD	get
PUT	update
PATCH	patch
DELETE	delete

资源和子资源根据传入请求的路径确定：

Kubelet API	资源	子资源
/stats/*	nodes	stats
/metrics/*	nodes	metrics
/logs/*	nodes	log
/spec/*	nodes	spec
all others	nodes	proxy

Namespace 和 API 组属性总是空字符串，资源的名字总是 kubelet 的 `Node` API 对象的名字。

当以该模式运行时，请确保用户为 apiserver 指定了 `--kubelet-client-certificate` 和 `--kubelet-client-key` 标志并授权了如下属性：

- verb=*, resource=nodes, subresource=proxy
- verb=*, resource=nodes, subresource=stats
- verb=*, resource=nodes, subresource=log
- verb=*, resource=nodes, subresource=spec
- verb=*, resource=nodes, subresource=metrics

TLS Bootstrap

本文档介绍如何为 kubelet 设置 TLS 客户端证书引导（bootstrap）。

Kubernetes 1.4 引入了一个用于从集群级证书颁发机构（CA）请求证书的 API。此 API 的原始目的是为 kubelet 提供 TLS 客户端证书。可以在 这里 找到该提议，在 feature #43 追踪该功能的进度。

kube-apiserver 配置

您必须提供一个 token 文件，该文件中指定了至少一个分配给 kubelet 特定 bootstrap 组的 “bootstrap token”。

该组将作为 controller manager 配置中的默认批准控制器而用于审批。随着此功能的成熟，您应该确保 token 被绑定到基于角色的访问控制（RBAC）策略上，该策略严格限制了与证书配置相关的客户端请求（使用 bootstrap token）。使用 RBAC，将 token 范围划分为组可以带来很大的灵活性（例如，当您配置完成节点后，您可以禁用特定引导组的访问）。

Token 认证文件

Token 可以是任意的，但应该可以表示为从安全随机数生成器（例如大多数现代操作系统中的 /dev/urandom）导出的至少128位熵。生成 token 有很多中方式。例如：

`head -c 16 /dev/urandom | od -An -t x | tr -d ' '`

产生的 token 类似于这样： `02b50b05283e98dd0fd71db496ef01e8` 。

Token 文件应该类似于以下示例，其中前三个值可以是任何值，引用的组名称应如下所示：

```
02b50b05283e98dd0fd71db496ef01e8,kubelet-bootstrap,10001,"system:kubelet-bootstrap"
```

在 kube-apiserver 命令中添加 `--token-auth-file=FILENAME` 标志（可能在您的 systemd unit 文件中）来启用 token 文件。

查看 该文档 获取更多详细信息。

客户端证书 CA 包

在 kube-apiserver 命令中添加 `--client-ca-file=FILENAME` 标志启用客户端证书认证，指定包含签名证书的证书颁发机构包（例如 `--client-ca-file=/var/lib/kubernetes/ca.pem`）。

kube-controller-manager 配置

请求证书的 API 向 Kubernetes controller manager 中添加证书颁发控制循环。使用磁盘上的 cfssl 本地签名文件的形式。目前，所有发型的证书均为一年有效期和并具有一系列关键用途。

签名文件

您必须提供证书颁发机构，这样才能提供颁发证书所需的密码资料。

kube-apiserver 通过指定的 `--client-ca-file=FILENAME` 标志来认证和采信该 CA。CA 的管理超出了本文档的范围，但建议您为 Kubernetes 生成专用的 CA。

假定证书和密钥都是 PEM 编码的。

Kube-controller-manager 标志为：

```
--cluster-signing-cert-file="/etc/path/to/kubernetes/ca/ca.crt"
--cluster-signing-key-file="/etc/path/to/kubernetes/ca/ca.key"
```

审批控制器

在 kubernetes 1.7 版本中，实验性的“组自动批准”控制器被弃用，新的 `csrapproving` 控制器将作为 kube-controller-manager 的一部分，被默认启用。

控制器使用 `SubjectAccessReview` API 来确定给定用户是否已被授权允许请求 CSR，然后根据授权结果进行批准。为了防止与其他批准者冲突，内置审批者没有明确地拒绝 CSR，只是忽略未经授权的请求。

控制器将 CSR 分为三个子资源：

1. `nodeclient` ：用户的客户端认证请求 `O=system:nodes` ，`CN=system:node:(node name)` 。
2. `selfnodeclient` ：更新具有相同 `O` 和 `CN` 的客户端证书的节点。
3. `selfnodeserver` ：更新服务证书的节点（ALPHA，需要 feature gate）。

当前，确定 CSR 是否为 `selfnodeserver` 请求的检查与 kubelet 的凭据轮换实现（Alpha 功能）相关联。因此，`selfnodeserver` 的定义将来可能会改变，并且需要 Controller Manager 上的 `RotateKubeletServerCertificate` feature gate。该功能的进展可以在 kubernetes/feature/#267 上追踪。

```
--feature-gates=RotateKubeletServerCertificate=true
```

以下 RBAC `ClusterRoles` 代表 `nodeClient` 、`selfnodeclient` 和 `selfnodeserver` 功能。在以后的版本中可能会自动创建类似的角色。

```
# A ClusterRole which instructs the CSR approver to approve a us
er requesting
# node client credentials.
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1beta1
metadata:
  name: approve-node-client-csr
rules:
- apiGroups: ["certificates.k8s.io"]
  resources: ["certificatesigningrequests/nodeclient"]
  verbs: ["create"]
---
# A ClusterRole which instructs the CSR approver to approve a no
de renewing its
# own client credentials.
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1beta1
metadata:
  name: approve-node-client-renewal-csr
rules:
- apiGroups: ["certificates.k8s.io"]
  resources: ["certificatesigningrequests/selfnodeclient"]
  verbs: ["create"]
---
# A ClusterRole which instructs the CSR approver to approve a no
de requesting a
# serving cert matching its client cert.
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1beta1
metadata:
  name: approve-node-server-renewal-csr
rules:
- apiGroups: ["certificates.k8s.io"]
  resources: ["certificatesigningrequests/selfnodeserver"]
  verbs: ["create"]
```

这些权力可以授予给凭证，如 bootstrap token。例如，要复制由已被移除的自动批准标志提供的行为，由单个组批准所有的 CSR：

```
# REMOVED: This flag no longer works as of 1.7.
--insecure-experimental-approve-all-kubelet-csrs-for-group="kube
let-bootstrap-token"
```

管理员将创建一个 `ClusterRoleBinding` 来定位该组。

```
# Approve all CSRs for the group "kubelet-bootstrap-token"
kind: ClusterRoleBinding
apiVersion: rbac.authorization.k8s.io/v1beta1
metadata:
  name: auto-approve-csrs-for-group
subjects:
- kind: Group
  name: kubelet-bootstrap-token
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: ClusterRole
  name: approve-node-client-csr
  apiGroup: rbac.authorization.k8s.io
```

要让节点更新自己的凭据，管理员可以构造一个 `ClusterRoleBinding` 来定位该节点的凭据。

```
kind: ClusterRoleBinding
apiVersion: rbac.authorization.k8s.io/v1beta1
metadata:
  name: node1-client-cert-renewal
subjects:
- kind: User
  name: system:node:node-1 # Let "node-1" renew its client certi
ficate.
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: ClusterRole
  name: approve-node-client-renewal-csr
  apiGroup: rbac.authorization.k8s.io
```

删除该绑定将会阻止节点更新客户端凭据，一旦其证书到期，实际上就会将其从集群中删除。

kubelet 配置

要向 kube-apiserver 请求客户端证书，kubelet 首先需要一个包含 bootstrap 身份验证 token 的 kubeconfig 文件路径。您可以使用 `kubectl config set-cluster`，`set-credentials` 和 `set-context` 来构建此 kubeconfig 文件。为 `kubectl config set-credentials` 提供 `kubelet-bootstrap` 的名称，并包含 `--token = <token-value>`，如下所示：

```
kubectl config set-credentials kubelet-bootstrap --token=${BOOTSTRAP_TOKEN} --kubeconfig=bootstrap.kubeconfig
```

启动 kubelet 时，如果 `--kubeconfig` 指定的文件不存在，则使用 bootstrap kubeconfig 向 API server 请求客户端证书。在批准 `kubelet` 的证书请求和回执时，将包含了生成的密钥和证书的 kubeconfig 文件写入由 `-kubeconfig` 指定的路径。证书和密钥文件将被放置在由 `--cert-dir` 指定的目录中。

启动 kubelet 时启用 bootstrap 用到的标志：

```
--experimental-bootstrap-kubeconfig="/path/to/bootstrap/kubeconfig"
```

此外，在1.7中，kubelet 实现了 **Alpha** 功能，使其客户端和/或服务器都能轮转提供证书。

可以分别通过 kubelet 中的 `RotateKubeletClientCertificate` 和 `RotateKubeletServerCertificate` 功能标志启用此功能，但在未来版本中可能会以向后兼容的方式发生变化。

```
--feature-gates=RotateKubeletClientCertificate=true,RotateKubeletServerCertificate=true
```

`RotateKubeletClientCertificate` 可以让 kubelet 在其现有凭据到期时通过创建新的 CSR 来轮换其客户端证书。 `RotateKubeletServerCertificate` 可以让 kubelet 在其引导客户端凭据后还可以请求服务证书，并轮换该证书。服务证书目前不要求 DNS 或 IP SANs。

kubectl 审批

签名控制器不会立即签署所有证书请求。相反，它会一直等待直到适当特权的用户被标记为“已批准”状态。这最终将是由外部审批控制器来处理的自动化过程，但是对于 alpha 版本的 API 来说，可以由集群管理员通过 kubectl 命令手动完成。

管理员可以使用 `kubectl get csr` 命令列出所有的 CSR，使用 `kubectl describe csr <name>` 命令描述某个 CSR的详细信息。在 1.6 版本以前，没有直接的批准/拒绝命令 ，因此审批者需要直接更新 Status 信息（查看如何实现）。此

后的 Kubernetes 版本中提供了 `kubectl certificate approve <name>` 和 `kubectl certificate deny <name>` 命令。

创建用户认证授权的kubeconfig文件

当我们安装好集群后，如果想要把 kubectl 命令交给用户使用，就不得不对用户的身份进行认证和对其权限做出限制。

下面以创建一个 devuser 用户并将其绑定到 dev 和 test 两个 namespace 为例说明。

创建 CA 证书和秘钥

创建 `devuser-csr.json` 文件

```
{
  "CN": "devuser",
  "hosts": [],
  "key": {
    "algo": "rsa",
    "size": 2048
  },
  "names": [
    {
      "C": "CN",
      "ST": "BeiJing",
      "L": "BeiJing",
      "O": "k8s",
      "OU": "System"
    }
  ]
}
```

生成 **CA** 证书和私钥

在 创建 TLS 证书和秘钥 一节中我们将生成的证书和秘钥放在了所有节点的 `/etc/kubernetes/ssl` 目录下，下面我们再在 master 节点上为 devuser 创建证书和秘钥，在 `/etc/kubernetes/ssl` 目录下执行以下命令：

执行该命令前请先确保该目录下已经包含如下文件：

```
ca-key.pem  ca.pem ca-config.json  devuser-csr.json
```

```
$ cfssl gencert -ca=ca.pem -ca-key=ca-key.pem -config=ca-config.
json -profile=kubernetes devuser-csr.json | cfssljson -bare devu
ser
2017/08/31 13:31:54 [INFO] generate received request
2017/08/31 13:31:54 [INFO] received CSR
2017/08/31 13:31:54 [INFO] generating key: rsa-2048
2017/08/31 13:31:55 [INFO] encoded CSR
2017/08/31 13:31:55 [INFO] signed certificate with serial number
 43372632012323103879829229080989286813242051309
2017/08/31 13:31:55 [WARNING] This certificate lacks a "hosts" f
ield. This makes it unsuitable for
websites. For more information see the Baseline Requirements for
 the Issuance and Management
of Publicly-Trusted Certificates, v.1.1.6, from the CA/Browser F
orum (https://cabforum.org);
specifically, section 10.2.3 ("Information Requirements").
```

这将生成如下文件：

```
devuser.csr  devuser-key.pem  devuser.pem
```

创建 kubeconfig 文件

```
# 设置集群参数
export KUBE_APISERVER="https://172.20.0.113:6443"
kubectl config set-cluster kubernetes \
--certificate-authority=/etc/kubernetes/ssl/ca.pem \
--embed-certs=true \
--server=${KUBE_APISERVER} \
--kubeconfig=devuser.kubeconfig

# 设置客户端认证参数
kubectl config set-credentials devuser \
--client-certificate=/etc/kubernetes/ssl/devuser.pem \
--client-key=/etc/kubernetes/ssl/devuser-key.pem \
--embed-certs=true \
--kubeconfig=devuser.kubeconfig

# 设置上下文参数
kubectl config set-context kubernetes \
--cluster=kubernetes \
--user=devuser \
--namespace=dev \
--kubeconfig=devuser.kubeconfig

# 设置默认上下文
kubectl config use-context kubernetes --kubeconfig=devuser.kubec
onfig
```

我们现在查看 kubectl 的 context：

```
kubectl config get-contexts
CURRENT   NAME                 CLUSTER              AUTHINFO          NA
MESPACE
*         kubernetes           kubernetes           admin
          default-context      default-cluster      default-admin
```

显示的用户仍然是 admin，这是因为 kubectl 使用了 `$HOME/.kube/config` 文件作为了默认的 context 配置，我们只需要将其用刚生成的 `devuser.kubeconfig` 文件替换即可。

```
cp -f ./devuser.kubeconfig /root/.kube/config
```

关于 kubeconfig 文件的更多信息请参考 使用 kubeconfig 文件配置跨集群认证。

RoleBinding

如果我们想限制 devuser 用户的行为，需要使用 RBAC创建角色绑定以将该用户的行为限制在某个或某几个 namespace 空间范围内，例如：

```
kubectl create rolebinding devuser-admin-binding --clusterrole=a
dmin --user=devuser --namespace=dev
kubectl create rolebinding devuser-admin-binding --clusterrole=a
dmin --user=devuser --namespace=test
```

这样 devuser 用户对 dev 和 test 两个 namespace 具有完全访问权限。

让我们来验证以下，现在我们在执行：

```
# 获取当前的 context
kubectl config get-contexts
CURRENT   NAME         CLUSTER      AUTHINFO    NAMESPACE
*         kubernetes   kubernetes   devuser     dev
*         kubernetes   kubernetes   devuser     test

# 无法访问 default namespace
kubectl get pods --namespace default
Error from server (Forbidden): User "devuser" cannot list pods in
 the namespace "default". (get pods)

# 默认访问的是 dev namespace，您也可以重新设置 context 让其默认访问 tes
t namespace
kubectl get pods
No resources found.
```

现在 kubectl 命令默认使用的 context 就是 devuser 了，且该用户只能操作 dev 和 test 这两个 namespace，并拥有完全的访问权限。

可以使用我写的create-user.sh脚本来创建namespace和用户并授权，参考说明。

关于角色绑定的更多信息请参考 RBAC——基于角色的访问控制。

IP 伪装代理

本文将讲述如何配置和启用 ip-masq-agent。

创建 ip-masq-agent

要创建 ip-masq-agent，运行下面的 kubectl 命令：

```
kubectl create -f https://raw.githubusercontent.com/kubernetes-incubator/ip-masq-agent/master/ip-masq-agent.yaml
```

关于 ip-masq-agent 的更多信息请参考 该文档。

在大多数情况下，默认的一套规则应该是足够的；但是，如果内置的规则不适用于您的集群，您可以创建并应用 ConfigMap 来自定义受影响的 IP 范围。例如，为了仅允许 ip-masq-agent 考虑 10.0.0.0/8，您可以在名为 “config” 的文件中创建以下 ConfigMap。

```
nonMasqueradeCIDRs:
  - 10.0.0.0/8
resyncInterval: 60s
```

注意：重要的是，该文件被命名为 config，因为默认情况下，该文件将被用作 ip-masq-agent 查找的关键字。

运行下列命令将 ConfigMap 添加到您的集群中：

```
kubectl create configmap ip-masq-agent --from-file=config --namespace=kube-system
```

这将会更新 */etc/config/ip-masq-agent* 文件，并每隔 *resyscInterval* 时间段检查一遍该文件，将配置应用到集群的节点中。

```
iptables -t nat -L IP-MASQ-AGENT
Chain IP-MASQ-AGENT (1 references)
target     prot opt source               destination
RETURN     all  --  anywhere             169.254.0.0/16       /*
 ip-masq-agent: cluster-local traffic should not be subject to M
ASQUERADE */ ADDRTYPE match dst-type !LOCAL
RETURN     all  --  anywhere             10.0.0.0/8           /*
 ip-masq-agent: cluster-local
MASQUERADE  all  --  anywhere             anywhere             /
* ip-masq-agent: outbound traffic should be subject to MASQUERAD
E (this match must come after cluster-local CIDR matches) */ ADD
RTYPE match dst-type !LOCAL
```

默认情况下，本地链路范围（169.254.0.0/16）也由 ip-masq 代理处理，该代理设置相应的 iptables 规则。想要让 ip-masq-agent 忽略本地链路，您可以在 ConfigMap 中将 masqLinkLocal 设置为 true。

```
nonMasqueradeCIDRs:
  - 10.0.0.0/8
resyncInterval: 60s
masqLinkLocal: true
```

IP 伪装代理用户指南

ip-masq-agent 用户配置 iptables 规则将 Pod 的 IP 地址隐藏在集群 node 节点的 IP 地址后面。这通常在将流量发送到群集的 pod CIDR 范围之外的目的地时执行。

关键术语

- **NAT**（网络地址转换）

  是一种通过修改 IP 头中的源和/或目标地址信息来将一个 IP 地址重映射到另一个 IP 地址的方法。通常由执行 IP 路由的设备完成。

- **Masquerading**（伪装）

  NAT 的一种形式，通常用于执行多个地址转换，其中多个源 IP 地址被掩盖在单个地址之后，通常是由某设备进行 IP 路由。在 kubernetes 中，这是 Node 的 IP 地址。

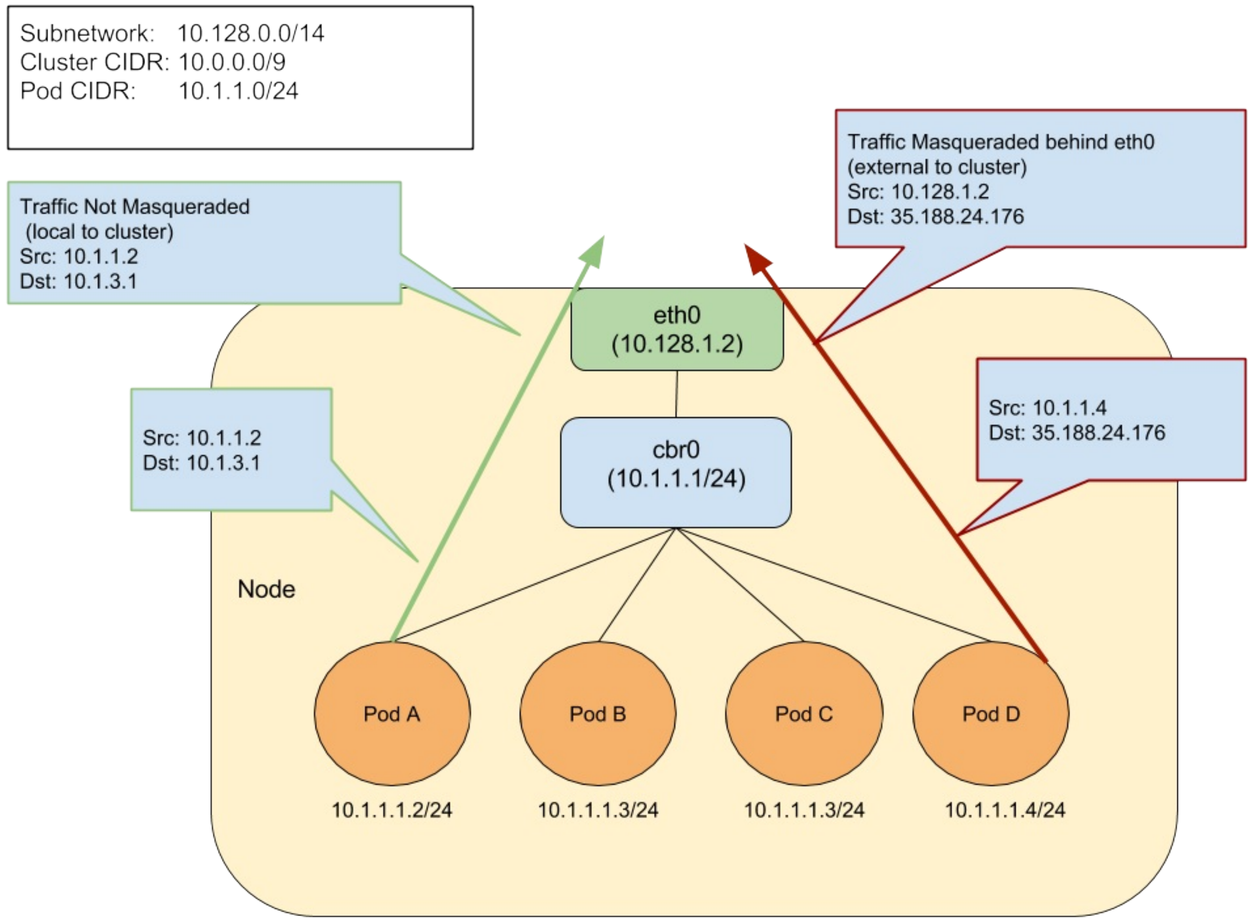
- **CIDR**（无类域内路由选择）

基于可变长度子网掩码，允许指定任意长度的前缀。CIDR 引入了一种新的 IP 地址表示方法，现在通常被称为 **CIDR** 表示法，将地址或路由前缀的比特位数作为后缀，例如 192.168.2.0/24。

- 本地链路

  本地链路地址是仅能在主机连接的网段或广播域内进行有效通信的网络地址。IPv4 的链路本地地址在 CIDR 表示法定义的地址块是 169.254.0.0/16。

Ip-masq-agent 在将流量发送到集群 node 节点的 IP 和 Cluster IP 范围之外的目的地时，会配置 iptables 规则来处理伪装的 node/pod IP 地址。这基本上将 pod 的 IP 地址隐藏在了集群 node 节点的 IP 地址后面。在某些环境中，到“外部”地址的流量必须来自已知的机器地址。例如，在 Google Cloud 中，到互联网的任何流量必须来自虚拟机的 IP。当使用容器时，如在GKE中，Pod IP 将被拒绝作为出口。为了避免这种情况，我们必须将 Pod IP 隐藏在 VM 自己的 IP 地址之后——通常被称为“伪装”。默认情况下，配置代理将 RFC 1918指定的三个专用 IP 范围视为非伪装 CIDR。范围包括 10.0.0.0/8、172.16.0.0/12 和 192.168.0.0/16。默认情况下，代理还将本地链路（169.254.0.0/16）视为非伪装 CIDR。代理配置为每隔60秒从 */etc/config/ip-masq-agent* 位置重新加载其配置，这也是可配置的。



图片 - *IP*伪装代理示意图

代理的配置文件必须使用 yaml 或 json 语法，并且包含以下三个可选的 key：

- **nonMasqueradeCIDRs**：使用 CIDR 表示法指定的非伪装范围的字符串列表。
- **masqLinkLocal**：一个布尔值（true/false），表示是否将流量伪装成本地链路前缀 169.254.0.0/16。默认为false。
- **resyncInterval**：代理尝试从磁盘重新加载配置的时间间隔。例如 '30s' 其中 's' 是秒，'ms' 是毫秒等...

到 10.0.0.0/8、172.16.0.0/12 和 192.168.0.0/16 范围的流量将不会被伪装。任何其他流量（假定是互联网）将被伪装。这里有个例子，来自 pod 的本地目的地址可以是其节点的 IP 地址、其他节点的地址或 Cluster IP 范围中的一个 IP 地址。其他任何流量都将默认伪装。以下条目显示 ip-masq-agent 应用的默认规则集：

```
iptables -t nat -L IP-MASQ-AGENT
RETURN     all  --  anywhere             169.254.0.0/16       /*
 ip-masq-agent: cluster-local traffic should not be subject to M
ASQUERADE */ ADDRTYPE match dst-type !LOCAL
RETURN     all  --  anywhere             10.0.0.0/8           /*
 ip-masq-agent: cluster-local traffic should not be subject to M
ASQUERADE */ ADDRTYPE match dst-type !LOCAL
RETURN     all  --  anywhere             172.16.0.0/12        /*
 ip-masq-agent: cluster-local traffic should not be subject to M
ASQUERADE */ ADDRTYPE match dst-type !LOCAL
RETURN     all  --  anywhere             192.168.0.0/16       /*
 ip-masq-agent: cluster-local traffic should not be subject to M
ASQUERADE */ ADDRTYPE match dst-type !LOCAL
MASQUERADE  all  --  anywhere             anywhere             /
* ip-masq-agent: outbound traffic should be subject to MASQUERAD
E (this match must come after cluster-local CIDR matches) */ ADD
RTYPE match dst-type !LOCAL
```

默认情况下，在 GCE/GKE 中将启动 kubernetes 1.7.0 版本，ip-masq-agent 已经在集群中运行。如果您在其他环境中运行 kubernetes，那么您可以将 ip-masq-agent 以 DaemonSet 的方式在集群中运行。

原文地址：https://k8smeetup.github.io/docs/tasks/administer-cluster/ip-masq-agent/

译者：rootsongjc

使用 kubeconfig 或 token 进行用户身份认证

在开启了 TLS 的集群中，每当与集群交互的时候少不了的是身份认证，使用 kubeconfig（即证书） 和 token 两种认证方式是最简单也最通用的认证方式，在 dashboard 的登录功能就可以使用这两种登录功能。

下文分两块以示例的方式来讲解两种登陆认证方式：

- 为 brand 命名空间下的 brand 用户创建 kubeconfig 文件
- 为集群的管理员（拥有所有命名空间的 amdin 权限）创建 token

使用 kubeconfig

如何生成 `kubeconfig` 文件请参考创建用户认证授权的kubeconfig文件。

> 注意我们生成的 kubeconfig 文件中没有 token 字段，需要手动添加该字段。

比如我们为 brand namespace 下的 brand 用户生成了名为 `brand.kubeconfig` 的 kubeconfig 文件，还要再该文件中追加一行 `token` 的配置（如何生成 token 将在下文介绍），如下所示：

```
brand.kubeconfig
apiVersion: v1
clusters:
- cluster:
    certificate-authority-data: LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSURtakN
    server: https://172.20.0.113:6443
  name: kubernetes
contexts:
- context:
    cluster: kubernetes
    namespace: brand
    user: brand
  name: kubernetes
current-context: "kubernetes"
kind: Config
preferences: {}
users:
- name: brand
  user:
    client-certificate-data: LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUQwakNDQX
    client-key-data: LS0tLS1CRUdJTiBSU0EgUFJJVkFURSBLRVktLS0tLQpNSUlFb3dJQkFBS0
    token: a09bb459d67d876cf1829b4047394a5a
```

图片 - *kubeconfig*文件

对于访问 dashboard 时候的使用 kubeconfig 文件如 `brand.kubeconfig` 必须追到 `token` 字段，否则认证不会通过。而使用 kubectl 命令时的用的 kubeconfig 文件则不需要包含 `token` 字段。

生成 token

需要创建一个admin用户并授予admin角色绑定，使用下面的yaml文件创建admin用户并赋予他管理员权限，然后可以通过token访问kubernetes，该文件见admin-role.yaml。

生成kubernetes集群最高权限admin用户的token

```yaml
kind: ClusterRoleBinding
apiVersion: rbac.authorization.k8s.io/v1beta1
metadata:
  name: admin
  annotations:
    rbac.authorization.kubernetes.io/autoupdate: "true"
roleRef:
  kind: ClusterRole
  name: cluster-admin
  apiGroup: rbac.authorization.k8s.io
subjects:
- kind: ServiceAccount
  name: admin
  namespace: kube-system
---
apiVersion: v1
kind: ServiceAccount
metadata:
  name: admin
  namespace: kube-system
  labels:
    kubernetes.io/cluster-service: "true"
    addonmanager.kubernetes.io/mode: Reconcile
```

然后执行下面的命令创建 serviceaccount 和角色绑定，

```
kubectl create -f admin-role.yaml
```

创建完成后获取secret中token的值。

```
# 获取admin-token的secret名字
$ kubectl -n kube-system get secret|grep admin-token
admin-token-nwphb                          kubernetes.io/service
-account-token   3         6m
# 获取token的值
$ kubectl -n kube-system describe secret admin-token-nwphb
Name:         admin-token-nwphb
Namespace:     kube-system
Labels:         <none>
Annotations:    kubernetes.io/service-account.name=admin
        kubernetes.io/service-account.uid=f37bd044-bfb3-11e7-87c
0-f4e9d49f8ed0

Type:    kubernetes.io/service-account-token

Data
====
namespace:    11 bytes
token:        非常长的字符串
ca.crt:        1310 bytes
```

也可以使用 jsonpath 的方式直接获取 token 的值，如：

```
kubectl -n kube-system get secret admin-token-nwphb -o jsonpath={
.data.token}|base64 -d
```

注意：yaml 输出里的那个 token 值是进行 base64 编码后的结果，一定要将 kubectl 的输出中的 token 值进行 `base64` 解码，在线解码工具 base64decode，Linux 和 Mac 有自带的 `base64` 命令也可以直接使用，输入 `base64` 是进行编码，Linux 中 `base64 -d` 表示解码，Mac 中使用 `base64 -D` 。

我们使用了 base64 对其重新解码，因为 secret 都是经过 base64 编码的，如果直接使用 kubectl 中查看到的 `token` 值会认证失败，详见 secret 配置。关于 JSONPath 的使用请参考 JSONPath 手册。

更简单的方式是直接使用 `kubectl describe` 命令获取token的内容（经过base64解码之后）：

```
kubectl describe secret admin-token-nwphb
```

为普通用户生成token

为指定namespace分配该namespace的最高权限，这通常是在为某个用户（组织或者个人）划分了namespace之后，需要给该用户创建token登陆kubernetes dashboard或者调用kubernetes API的时候使用。

每次创建了新的namespace下都会生成一个默认的token，名为 `default-token-xxxx` 。 `default` 就相当于该namespace下的一个用户，可以使用下面的命令给该用户分配该namespace的管理员权限。

```
kubectl create rolebinding $ROLEBINDING_NAME --clusterrole=admin
 --serviceaccount=$NAMESPACE:default --namespace=$NAMESPACE
```

- `$ROLEBINDING_NAME` 必须是该namespace下的唯一的
- `admin` 表示用户该namespace的管理员权限，关于使用 `clusterrole` 进行更细粒度的权限控制请参考RBAC——基于角色的访问控制。
- 我们给默认的serviceaccount `default` 分配admin权限，这样就不要再创建新的serviceaccount，当然你也可以自己创建新的serviceaccount，然后给它admin权限

参考

- JSONPath 手册
- Kubernetes 中的认证

Kubernetes 中的用户与身份认证授权

在安装集群的时候我们在 master 节点上生成了一堆证书、token，还在 kubelet 的配置中用到了 bootstrap token，安装各种应用时，为了能够与 API server 通信创建了各种 service account，在 Dashboard 中使用了 kubeconfig 或 token 登陆，那么这些都属于什么认证方式？如何区分用户的？我特地翻译了下这篇官方文档，想你看了之后你将找到答案。

重点查看 bearer token 和 HTTP 认证中的 token 使用，我们已经有所应用，如 使用kubeconfig或token进行用户身份认证。

认识 Kubernetes 中的用户

Kubernetes 集群中包含两类用户：一类是由 Kubernetes 管理的 service account，另一类是普通用户。

普通用户被假定为由外部独立服务管理。管理员分发私钥，用户存储（如 Keystone 或 Google 帐户），甚至包含用户名和密码列表的文件。在这方面，*Kubernetes* 没有代表普通用户帐户的对象。无法通过 API 调用的方式向集群中添加普通用户。

相对的，service account 是由 Kubernetes API 管理的帐户。它们都绑定到了特定的 namespace，并由 API server 自动创建，或者通过 API 调用手动创建。Service account 关联了一套凭证，存储在 `Secret`，这些凭证同时被挂载到 pod 中，从而允许 pod 与 kubernetes API 之间的调用。

API 请求被绑定到普通用户或 serivce account 上，或者作为匿名请求对待。这意味着集群内部或外部的每个进程，无论从在工作站上输入 `kubectl` 的人类用户到节点上的 `kubelet`，到控制平面的成员，都必须在向 API Server 发出请求时进行身份验证，或者被视为匿名用户。

认证策略

Kubernetes 使用客户端证书、bearer token、身份验证代理或者 HTTP 基本身份验证等身份认证插件来对 API 请求进行身份验证。当有 HTTP 请求发送到 API server 时，插件会尝试将以下属性关联到请求上：

- 用户名：标识最终用户的字符串。常用值可能是 `kube-admin` 或 `jane@example.com`。
- UID：标识最终用户的字符串，比用户名更加一致且唯一。
- 组：一组将用户和常规用户组相关联的字符串。
- 额外字段：包含其他有用认证信息的字符串列表的映射。

所有的值对于认证系统都是不透明的，只有 授权人 才能解释这些值的重要含义。

您可以一次性启用多种身份验证方式。通常使用至少以下两种认证方式：

- 服务帐户的 service account token
- 至少一种其他的用户认证的方式

当启用了多个认证模块时，第一个认证模块成功认证后将短路请求，不会进行第二个模块的认证。API server 不会保证认证的顺序。

`system:authenticated` 组包含在所有已验证用户的组列表中。

与其他身份验证协议（LDAP、SAML、Kerberos、x509 方案等）的集成可以使用 身份验证代理 或 身份验证 webhook 来实现。

X509 客户端证书

通过将 `--client-ca-file=SOMEFILE` 选项传递给 API server 来启用客户端证书认证。引用的文件必须包含一个或多个证书颁发机构，用于验证提交给 API server 的客户端证书。如果客户端证书已提交并验证，则使用 subject 的 Common Name（CN）作为请求的用户名。从 Kubernetes 1.4开始，客户端证书还可以使用证书的 organization 字段来指示用户的组成员身份。要为用户包含多个组成员身份，请在证书中包含多个 organization 字段。

例如，使用 `openssl` 命令工具生成用于签名认证请求的证书：

```
openssl req -new -key jbeda.pem -out jbeda-csr.pem -subj "/CN=jbeda/O=app1/O=app2"
```

这将为一个用户名为 ”jbeda“ 的 CSR，属于两个组“app1”和“app2”。

如何生成客户端证书请参阅 附录。

静态 Token 文件

当在命令行上指定 `--token-auth-file=SOMEFILE` 选项时，API server 从文件读取 bearer token。目前，token 会无限期地持续下去，并且不重新启动 API server 的话就无法更改令牌列表。

token 文件是一个 csv 文件，每行至少包含三列：token、用户名、用户 uid，其次是可选的组名。请注意，如果您有多个组，则该列必须使用双引号。

```
token,user,uid,"group1,group2,group3"
```

在请求中放置 Bearer Token

当使用来自 http 客户端的 bearer token 时，API server 期望 `Authorization` header 中包含 `Bearer token` 的值。Bearer token 必须是一个字符串序列，只需使用 HTTP 的编码和引用功能就可以将其放入到 HTTP header 中。例如：如果 bearer token 是 `31ada4fd-adec-460c-809a-9e56ceb75269` ，那么它将出现在 HTTP header 中，如下所示：

```
Authorization: Bearer 31ada4fd-adec-460c-809a-9e56ceb75269
```

Bootstrap Token

该功能仍处于 **alpha** 版本。

为了简化新集群的初始化引导过程，Kubernetes 中包含了一个名为 *Bootstrap Token* 的动态管理的 bearer token。这些 token 使用 Secret 存储在 `kube-system` namespace 中，在那里它们可以被动态管理和创建。Controller Manager 中包含了一个 TokenCleaner 控制器，用于在 bootstrap token 过期时删除将其删除。

这些 token 的形式是 `[a-z0-9]{6}.[a-z0-9]{16}` 。第一部分是 Token ID，第二部分是 Token Secret。您在 HTTP header 中指定的 token 如下所示：

```
Authorization: Bearer 781292.db7bc3a58fc5f07e
```

在 API server 的启动参数中加上 `--experimental-bootstrap-token-auth` 标志以启用 Bootstrap Token Authenticator。您必须通过 Controller Manager 上的 `--controllers` 标志启用 TokenCleaner 控制器，如 `--controllers=*,tokencleaner`。如果您使用它来引导集群， `kubeadm` 会为您完成。

认证者认证为 `system:bootstrap:<Token ID>` 。被包含在 `system:bootstrappers` 组中。命名和组是有意限制用户使用过去的 bootstap token。可以使用用户名和组（ `kubeadm` 使用）来制定适当的授权策略以支持引导集群。

有关 Bootstrap Token 身份验证器和控制器的更深入的文档，以及如何使用 `kubeadm` 管理这些令牌，请参阅 Bootstrap Token。

静态密码文件

通过将 `--basic-auth-file=SOMEFILE` 选项传递给 API server 来启用基本身份验证。目前，基本身份验证凭证将无限期地保留，并且密码在不重新启动API服务器的情况下无法更改。请注意，为了方便起见，目前支持基本身份验证，而上述模式更安全更容易使用。

基本身份认证是一个 csv 文件，至少包含3列：密码、用户名和用户 ID。在 Kubernetes 1.6 和更高版本中，可以指定包含以逗号分隔的组名称的可选第四列。如果您有多个组，则必须将第四列值用双引号（“）括起来，请参阅以下示例：

```
password,user,uid,"group1,group2,group3"
```

当使用来自 HTTP 客户端的基本身份验证时，API server 需要 `Authorization` header 中包含 `Basic BASE64ENCODED(USER:PASSWORD)` 的值。

Service Account Token

Service account 是一个自动启用的验证器，它使用签名的 bearer token 来验证请求。该插件包括两个可选的标志：

- `--service-account-key-file` 一个包含签名 bearer token 的 PEM 编码文件。如果未指定，将使用 API server 的 TLS 私钥。
- `--service-account-lookup` 如果启用，从 API 中删除掉的 token 将被撤

销。

Service account 通常 API server 自动创建，并通过 `ServiceAccount` 注入控制器 关联到集群中运行的 Pod 上。Bearer token 挂载到 pod 中众所周知的位置，并允许集群进程与 API server 通信。 帐户可以使用 `PodSpec` 的 `serviceAccountName` 字段显式地与Pod关联。

注意： `serviceAccountName` 通常被省略，因为这会自动生成。

```yaml
apiVersion: apps/v1beta2
kind: Deployment
metadata:
  name: nginx-deployment
  namespace: default
spec:
  replicas: 3
  template:
    metadata:
    # ...
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
        serviceAccountName: bob-the-bot
```

Service account bearer token 在集群外使用也是完全有效的，并且可以用于为希望与 Kubernetes 通信的长期运行作业创建身份。要手动创建 service account，只需要使用 `kubectl create serviceaccount (NAME)` 命令。这将在当前的 namespace 和相关连的 secret 中创建一个 service account。

```
$ kubectl create serviceaccount jenkins
serviceaccount "jenkins" created
$ kubectl get serviceaccounts jenkins -o yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  # ...
secrets:
- name: jenkins-token-1yvwg
```

创建出的 secret 中拥有 API server 的公共 CA 和前面的 JSON Web Token（JWT）。

```
$ kubectl get secret jenkins-token-1yvwg -o yaml
apiVersion: v1
data:
  ca.crt: (APISERVER'S CA BASE64 ENCODED)
  namespace: ZGVmYXVsdA==
  token: (BEARER TOKEN BASE64 ENCODED)
kind: Secret
metadata:
  # ...
type: kubernetes.io/service-account-token
```

注意：所有值是基于 base64 编码的，因为 secret 总是基于 base64 编码。

经过签名的 JWT 可以用作 bearer token 与给定的 service account 进行身份验证。请参阅 上面 关于如何在请求中放置 bearer token。通常情况下，这些 secret 被挂载到 pod 中，以便对集群内的 API server 进行访问，但也可以从集群外访问。

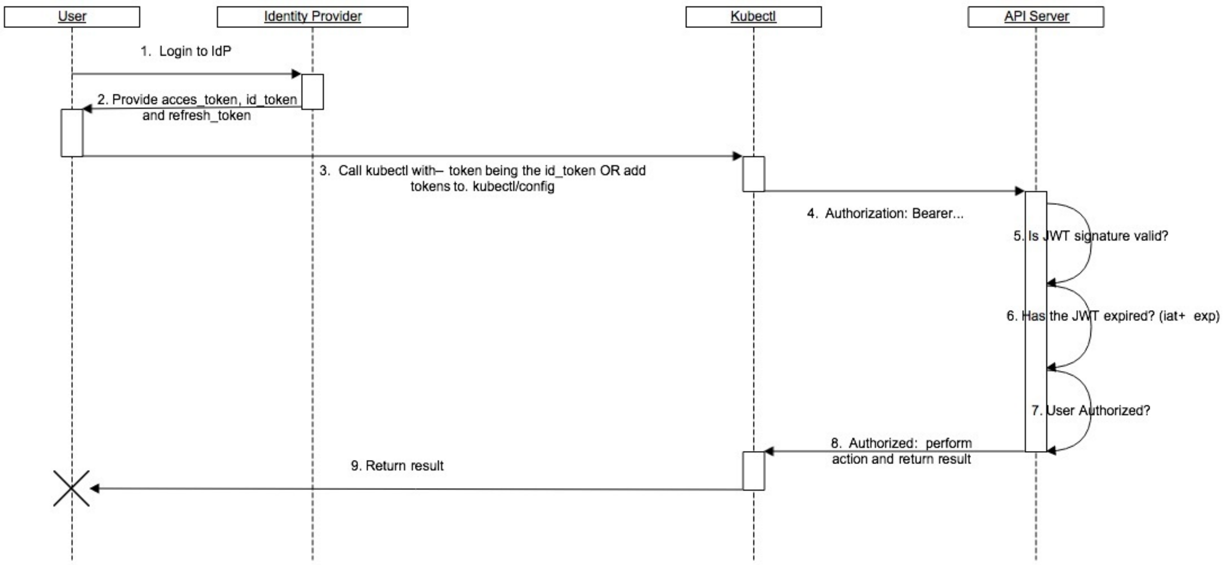
Service account 验证时用户名 `system:serviceaccount:(NAMESPACE):(SERVICEACCOUNT)` ，被指定到组 `system:serviceaccounts` 和 `system:serviceaccounts:(NAMESPACE)` 。

注意：由于 service account 的 token 存储在 secret 中，所以具有对这些 secret 的读取权限的任何用户都可以作为 service account 进行身份验证。授予 service account 权限和读取 secret 功能时要谨慎。

OpenID Connect Token

OpenID Connect 是由 OAuth2 供应商提供的 OAuth2，特别是 Azure Active Directory、Salesforce 和 Google。对 OAuth2 协议的主要扩展是返回一个称作 ID Token 的格外字段。该 token 是一个 JSON Web Token (JWT) ，有服务器签名，具有众所周知的字段，如用户的电子邮件。

为了识别用户，认证者使用 OAuth2 token 响应 中的 `id_token` （而不是 `access_token` ）作为 bearer token。请参阅 上面 的关于将 token 置于请求中。



图片 - *Kubernetes OpenID Connect Flow*

1. 登陆到您的身份提供商
2. 您的身份提供商将为您提供一个 `access_token`，一个 `id_token` 和一个 `refresh_token`
3. 当使用 `kubectl` 时，使用 `--token` 标志和 `id_token` ，或者直接加入到您的 `kubeconfig` 文件中
4. `kubectl` 在调用 API server 时将 `id_token` 置于 HTTP header 中
5. API server 将通过检查配置中指定的证书来确保 JWT 签名有效
6. 检查以确保 `id_token` 没有过期
7. 确保用户已授权
8. 授权 API server 后向 `kubectl`
9. `kubectl` 向用户提供反馈

由于所有需要验证您身份的数据都在 `id_token` 中，Kubernetes 不需要向身份提供商 “phone home”。在每个请求都是无状态的模型中，这为认证提供了非常可扩展的解决方案。它确实提供了一些挑战：

1. Kubernetes 没有 ”web 接口“ 来出发验证进程。没有浏览器或界面来收集凭据，这就是为什么您需要首先认证您的身份提供商。
2. `id_token` 无法撤销，就像一个证书，所以它应该是短暂的（只有几分钟），所以每隔几分钟就得到一个新的令牌是非常烦人的。
3. 没有使用 `kubectl proxy` 命令或注入 `id_token` 的反向代理，无法简单地对 Kubernetes dashboard 进行身份验证。

配置 API Server

要启用该插件，需要在 API server 中配置如下标志：

参数	描述	示例
`--oidc-issuer-url`	允许 API server 发现公共签名密钥的提供者的 URL。只接受使用 `https://` 的方案。通常是提供商的 URL 地址，不包含路径，例如“https://accounts.google.com” 或者 “https://login.salesforce.com”。这个 URL 应该指向下面的 .well-known/openid-configuration	如果发现 URL 是 `https://accounts.google.c` `known/openid-configuratior` 是 `https://accounts.google`
`--oidc-client-id`	所有的 token 必须为其颁发的客户端 ID	kubernetes
`--oidc-username-claim`	JWT声明使用的用户名。默认情况下， `sub` 是最终用户的唯一标识符。管理员可以选择其他声明，如 `email` 或 `name` ，具体取决于他们的提供者。不过， `email` 以外的其他声明将以发行者的 URL 作为前缀，以防止与其他插件命名冲突。	sub
`--oidc-groups-claim`	JWT声明使用的用户组。如果生命存在，它必须是一个字符串数组。	groups
`--oidc-ca-file`	用来签名您的身份提供商的网络 CA 证书的路径。默认为主机的跟 CA。	`/etc/kubernetes/ssl/kc-ca`

如果为 `--oidc-username-claim` 选择了除 `email` 以外的其他声明，则该值将以 `--oidc-issuer-url` 作为前缀，以防止与现有 Kubernetes 名称（例如 `system:users` ）冲突。例如，如果提供商网址是 https://accounts.google.com，而用户名声明映射到 `jane` ，则插件会将用户身份验证为：

```
https://accounts.google.com#jane
```

重要的是，API server 不是 OAuth2 客户端，而只能配置为信任单个发行者。这允许使用 Google 等公共提供者，而不必信任第三方发行的凭据。希望利用多个 OAuth 客户端的管理员应该探索支持 `azp`（授权方）声明的提供者，这是允许一个客户端代表另一个客户端发放令牌的机制。

Kubernetes不提供 OpenID Connect 身份提供商。您可以使用现有的公共 OpenID Connect 标识提供程序（例如Google 或 其他）。或者，您可以运行自己的身份提供程序，例如 CoreOS dex、Keycloak、CloudFoundry UAA 或 Tremolo Security 的 OpenUnison。

对于身份提供商能够适用于 Kubernetes，必须满足如下条件：Kubernetes it must:

1. 支持 OpenID connect 发现；不必是全部。
2. 使用非过时密码在TLS中运行
3. 拥有 CA 签名证书（即使 CA 不是商业 CA 或自签名）

有关上述要求3的说明，需要 CA 签名证书。如果您部署自己的身份提供商（而不是像 Google 或 Microsoft 之类的云提供商），则必须让您的身份提供商的 Web 服务器证书由 CA 标志设置为 TRUE 的证书签名，即使是自签名的。这是由于 GoLang 的 TLS 客户端实现对证书验证的标准非常严格。如果您没有 `CA`，可以使用 `CoreOS` 团队的 这个脚本 创建一个简单的 CA 和一个签名的证书和密钥对。

或者你可以使用 这个类似的脚本 来生成更长寿命和更长的 SHA256 证书密钥。

针对特定系统的安装说明：

- UAA
- Dex
- OpenUnison

使用 kubectl

选项 1 - OIDC 身份验证器

第一个选项是使用 `oidc` 身份验证器。此身份验证程序将您的 `id_token`、`refresh_token` 和您的 OIDC `client_secret` 自动刷新您的 token。一旦您对身份提供者进行了身份验证：

```
kubectl config set-credentials USER_NAME \
   --auth-provider=oidc \
   --auth-provider-arg=idp-issuer-url=( issuer url ) \
   --auth-provider-arg=client-id=( your client id ) \
   --auth-provider-arg=client-secret=( your client secret ) \
   --auth-provider-arg=refresh-token=( your refresh token ) \
   --auth-provider-arg=idp-certificate-authority=( path to your
ca certificate ) \
   --auth-provider-arg=id-token=( your id_token ) \
   --auth-provider-arg=extra-scopes=( comma separated list of sc
opes to add to "openid email profile", optional )
```

例如，在向身份提供者进行身份验证之后运行以下命令：

```
kubectl config set-credentials mmosley  \
        --auth-provider=oidc  \
        --auth-provider-arg=idp-issuer-url=https://oidcidp.tremo
lo.lan:8443/auth/idp/OidcIdP  \
        --auth-provider-arg=client-id=kubernetes  \
        --auth-provider-arg=client-secret=1db158f6-177d-4d9c-8a8
b-d36869918ec5  \
        --auth-provider-arg=refresh-token=q1bKLFOyUiosTfawzA93Tz
ZIDzH2TNa2SMm0zEiPKTUwME6BkEo6Sql5yUWVBSWpKUGphaWpxSVAfekBOZbBha
EW+VlFUeVRGcluyVF5JT4+haZmPsluFoFu5XkpXk5BXqHega4GAXlF+ma+vmYpFc
He5eZR+slBFpZKtQA= \
        --auth-provider-arg=idp-certificate-authority=/root/ca.p
em \
        --auth-provider-arg=extra-scopes=groups \
        --auth-provider-arg=id-token=eyJraWQiOiJDTj1vaWRjaWRwLnR
yZW1vbG8ubGFuLCBPVT1EZW1vLCBPPVRybWVvbG8gU2VjdXJpdHksIEw9QXJsaW5
ndG9uLCBTVD1WaXJnaW5pYSwgQz1VUy1DTj1rdWJlLWNhLTEyMDIxNDc5MjEwMzY
wNzMyMTUyIiwiYWxnIjoiUlMyNTYifQ.eyJpc3MiOiJodHRwczovL29pZGNpZHAu
dHJlbW9sby5sYW46ODQ0My9hdXRoL2lkcC9PaWRjSWRQIiwiYXVkIjoia3ViZXJu
ZXRlcyIsImV4cCI6MTQ4MzU0OTUxMSwianRpIjoiMm96US15TXdFcHV4WDlHZUhQ
dy1hZyIsImlhdCI6MTQ4MzU0OTQ1MSwibmJmIjoxNDgzNTQ5MzMxLCJzdWIiOiI0
YWViMzdiYS1iNjQ1LTQ4ZmQtYWIzMC0xYTAxZWU0MWUyMTgifQ.w6p4J_6qQ1HzT
G9nrEOrubxIMb9K5hzcMPxc9IxPx2K4xO9l-oFiUw93daH3m5pluP6K7eOE6txBu
RVfEcpJSwlelsOsW8gb8VJcnzMS9EnZpeA0tW_p-mnkFc3VcfyXuhe5R3G7aa5d8
uHv70yJ9Y3-UhjiN9EhpMdfPAoEB9fYKKkJRzF7utTTIPGrSaSU6d2pcpfYKaxIw
ePzEkT4DfcQthoZdy9ucNvvLoi1DIC-UocFD8HLs8LYKEqSxQvOcvnThbObJ9af7
1EwmuE21fO5KzMW20KtAeget1gnldOosPtz1G5EwvaQ401-RPQzPGMVBld0_zMCA
wZttJ4knw
```

将产生下面的配置：

```
users:
- name: mmosley
  user:
    auth-provider:
      config:
        client-id: kubernetes
        client-secret: 1db158f6-177d-4d9c-8a8b-d36869918ec5
        extra-scopes: groups
        id-token: eyJraWQiOiJDTj1vaWRjaWRwLnRyZW1vbG8ubGFuLCBPVT
1EZW1vLCBPPVRybWVvbG8gU2VjdXJpdHksIEw9QXJsaW5ndG9uLCBTVD1WaXJnaW
5pYSwgQz1VUy1DTj1rdWJlLWNhLTEyMDIxNDc5MjEwMzYwNzMyMTUyIiwiYWxnIj
oiUlMyNTYifQ.eyJpc3MiOiJodHRwczovL29pZGNpZHAudHJlbW9sby5sYW46ODQ
0My9hdXRoL2lkcC9PaWRjSWRQIiwiYXVkIjoia3ViZXJuZXRlcyIsImV4cCI6MTQ
4MzU0OTUxMSwianRpIjoiMm96US15TXdFcHV4WDlHZUhQdy1hZyIsImlhdCI6MTQ
4MzU0OTQ1MSwibmJmIjoxNDgzNTQ5MzMxLCJzdWIiOiI0YWViMzdiYS1iNjQ1LTQ
4ZmQtYWIzMC0xYTAxZWU0MWUyMTgifQ.w6p4J_6qQ1HzTG9nrEOrubxIMb9K5hzc
MPxc9IxPx2K4xO9l-oFiUw93daH3m5pluP6K7eOE6txBuRVfEcpJSwlelsOsW8gb
8VJcnzMS9EnZpeA0tW_p-mnkFc3VcfyXuhe5R3G7aa5d8uHv70yJ9Y3-UhjiN9Eh
pMdfPAoEB9fYKKkJRzF7utTTIPGrSaSU6d2pcpfYKaxIwePzEkT4DfcQthoZdy9u
cNvvLoi1DIC-UocFD8HLs8LYKEqSxQvOcvnThbObJ9af71EwmuE21fO5KzMW20Kt
Aeget1gnldOosPtz1G5EwvaQ401-RPQzPGMVBld0_zMCAwZttJ4knw
        idp-certificate-authority: /root/ca.pem
        idp-issuer-url: https://oidcidp.tremolo.lan:8443/auth/id
p/OidcIdP
        refresh-token: q1bKLFOyUiosTfawzA93TzZIDzH2TNa2SMm0zEiPK
TUwME6BkEo6Sql5yUWVBSWpKUGphaWpxSVAfekBOZbBhaEW+VlFUeVRGcluyVF5J
T4+haZmPsluFoFu5XkpXk5BXq
      name: oidc
```

一旦您的 `id_token` 过期， `kubectl` 将使用 `refresh_token` 刷新 `id_token` ，然后在 `kube/.config` 文件的 `client_secret` 中存储 `id_token` 的值和 `refresh_token` 的新值。

选项 **2 -** 使用 **`--token`** 选项

可以在 `kubectl` 命令的 `--token` 选项中传入 token。简单的拷贝和复制 `id_token` 到该选项中：

```
kubectl --token=eyJhbGciOiJSUzI1NiJ9.eyJpc3MiOiJodHRwczovL21sYi5
0cmVtb2xvLmxhbjo4MDQzL2F1dGgvaWRwL29pZGMiLCJhdWQiOiJrdWJlcm5ldGV
zIiwiZXhwIjoxNDc0NTk2NjY5LCJqdGkiOiI2RDUzNXoxUEpFNjJOR3QxaWVyYm9
RIiwiaWF0IjoxNDc0NTk2MzY5LCJuYmYiOjE0NzQ1OTYyNDksInN1YiI6Im13aW5
kdSIsInVzZXJfcm9sZSI6WyJ1c2VycyIsIm5ldy1uYW1lc3BhY2Utdmlld2VyIl0
sImVtYWlsIjoibXdpbmR1QG5vbW9yZWplZGkuY29tIn0.f2As579n9VNoaKzoF-d
OQGmXkFKf1FMyNV0-va_B63jn-_n9LGSCca_6IVMP8pO-Zb4KvRqGyTP0r3HkHxY
y5c81AnIh8ijarruczl-TK_yF5akjSTHFZD-0gRzlevBDiH8Q79NAr-ky0P4iIXS
8lY9Vnjch5MF74Zx0c3alKJHJUnnpjIACByfF2SCaYzbWFMUNat-K1PaUk5-ujMB
G7yYnr95xD-63n8CO8teGUAAEMx6zRjzfhnhbzX-ajwZLGwGUBT4WqjMs70-6a7_
8gZmLZb2az1cZynkFRj2BaCkVT3A2RrjeEwZEtGXlMqKJ1_I2ulrOVsYx01_yD35
-rw get nodes
```

Webhook Token 认证

Webhook 认证是用来认证 bearer token 的 hook。

- `--authentication-token-webhook-config-file` 是一个用来描述如何访问远程 webhook 服务的 kubeconfig 文件。
- `--authentication-token-webhook-cache-ttl` 缓存身份验证策略的时间。默认为两分钟。

配置文件使用 kubeconfig 文件格式。文件中的 ”user“ 指的是 API server 的 webhook，”clusters“ 是指远程服务。见下面的例子：

```
# clusters refers to the remote service.
clusters:
  - name: name-of-remote-authn-service
    cluster:
      certificate-authority: /path/to/ca.pem         # CA for ve
rifying the remote service.
      server: https://authn.example.com/authenticate # URL of re
mote service to query. Must use 'https'.

# users refers to the API server's webhook configuration.
users:
  - name: name-of-api-server
    user:
      client-certificate: /path/to/cert.pem # cert for the webho
ok plugin to use
      client-key: /path/to/key.pem          # key matching the c
ert

# kubeconfig files require a context. Provide one for the API se
rver.
current-context: webhook
contexts:
- context:
    cluster: name-of-remote-authn-service
    user: name-of-api-sever
  name: webhook
```

当客户端尝试使用 bearer token 与API server 进行认证是，如 上 论述，认证 webhook 用饱含该 token 的对象查询远程服务。Kubernetes 不会挑战缺少该 header 的请求。

请注意，webhook API对象与其他 Kubernetes API 对象具有相同的 版本控制兼容性规则。实现者应该意识到 Beta 对象的宽松兼容性承诺，并检查请求的 “apiVersion” 字段以确保正确的反序列化。此外，API server 必须启用 `authentication.k8s.io/v1beta1` API 扩展组（ `--runtime config =authentication.k8s.io/v1beta1=true` ）。

The request body will be of the following format:

```
{
  "apiVersion": "authentication.k8s.io/v1beta1",
  "kind": "TokenReview",
  "spec": {
    "token": "(BEARERTOKEN)"
  }
}
```

预计远程服务将填写请求的 `status` 字段以指示登录成功。响应主体的 `spec` 字段被忽略，可以省略。成功验证后的 bearer token 将返回：

```
{
  "apiVersion": "authentication.k8s.io/v1beta1",
  "kind": "TokenReview",
  "status": {
    "authenticated": true,
    "user": {
      "username": "janedoe@example.com",
      "uid": "42",
      "groups": [
        "developers",
        "qa"
      ],
      "extra": {
        "extrafield1": [
          "extravalue1",
          "extravalue2"
        ]
      }
    }
  }
}
```

未成功的请求将返回：

```
{
  "apiVersion": "authentication.k8s.io/v1beta1",
  "kind": "TokenReview",
  "status": {
    "authenticated": false
  }
}
```

HTTP状态代码可以用来提供额外的错误上下文。

认证代理

可以配置 API server 从请求 header 的值中识别用户，例如 `X-Remote-User` 。这样的设计是用来与请求 header 值的验证代理结合使用。

- `--requestheader-username-headers` 必需，大小写敏感。按 header 名称和顺序检查用户标识。包含值的第一个 header 将被作为用户名。

- `--requestheader-group-headers` 1.6 以上版本。可选。大小写敏感。建议为 “X-Remote-Group”。按 header 名称和顺序检查用户组。所有指定的 header 中的所有值都将作为组名。
- `--requestheader-extra-headers-prefix` 1.6 以上版本。可选，大小写敏感。建议为 “X-Remote-Extra-”。标题前缀可用于查找有关用户的额外信息（通常由配置的授权插件使用）。以任何指定的前缀开头的 header 都会删除前缀，header 名称的其余部分将成为额外的键值，而 header 值则是额外的值。

例如下面的配置：

```
--requestheader-username-headers=X-Remote-User
--requestheader-group-headers=X-Remote-Group
--requestheader-extra-headers-prefix=X-Remote-Extra-
```

该请求：

```
GET / HTTP/1.1
X-Remote-User: fido
X-Remote-Group: dogs
X-Remote-Group: dachshunds
X-Remote-Extra-Scopes: openid
X-Remote-Extra-Scopes: profile
```

将产生如下的用户信息：

```yaml
name: fido
groups:
- dogs
- dachshunds
extra:
  scopes:
  - openid
  - profile
```

为了防止 header 欺骗，验证代理需要在验证请求 header 之前向 API server 提供有效的客户端证书，以对照指定的 CA 进行验证。

- `--requestheader-client-ca-file` 必需。PEM 编码的证书包。在检查用户名的请求 header 之前，必须针对指定文件中的证书颁发机构提交并验证有效的客户端证书。

- `--requestheader-allowed-names` 可选。Common Name （cn）列表。如果设置了，则在检查用户名的请求 header 之前， 必须提供指定列表中 Common Name（cn）的有效客户端证书。如果为空，则允许使用任何 Common Name。

Keystone 密码

通过在启动过程中将 `--experimental-keystone-url=<AuthURL>` 选项传递给 API server 来启用 Keystone 认证。该插件在 `plugin/pkg/auth/authenticator/password/keystone/keystone.go` 中实现，目前使用基本身份验证通过用户名和密码验证用户。

如果您为 Keystone 服务器配置了自签名证书，则在启动 Kubernetes API server 时可能需要设置 `--experimental-keystone-ca-file=SOMEFILE` 选项。如果您设置了该选项，Keystone 服务器的证书将由 `experimental-keystone-ca-file` 中的某个权威机构验证。否则，证书由主机的根证书颁发机构验证。

有关如何使用 keystone 来管理项目和用户的详细信息，请参阅 Keystone 文档。请注意，这个插件仍处于试验阶段，正在积极开发之中，并可能在后续版本中进行更改。

请参考 讨论、蓝图 和 提出的改变 获取更多信息。

匿名请求

启用时，未被其他已配置身份验证方法拒绝的请求将被视为匿名请求，并给予 `system:anonymous` 的用户名和 `system:unuthenticated` 的组名。

例如，在配置了令牌认证和启用了匿名访问的服务器上，提供无效的 bearer token 的请求将收到 `401 Unauthorized` 错误。提供 bearer token 的请求将被视为匿名请求。

在 1.5.1 - 1.5.x 版本中，默认情况下命名访问是被禁用的，可以通过传递 `--anonymous-auth=false` 选项给 API server 来启用。

在 1.6+ 版本中，如果使用 `AlwaysAllow` 以外的授权模式，则默认启用匿名访问，并且可以通过将 `--anonymous-auth=false` 选项传递给API服务器来禁用。从 1.6 开始，ABAC 和 RBAC 授权人需要明确授权 `system:annoymous` 或

`system:unauthenticated` 组，因此授予对 `*` 用户或 `*` 组访问权的传统策略规则不包括匿名用户。

用户模拟

用户可以通过模拟 header 充当另一个用户。该请求会覆盖请求认证的用户信息。例如，管理员可以使用此功能通过暂时模拟其他用户并查看请求是否被拒绝来调试授权策略。

模拟请求首先认证为请求用户，然后切换到模拟的用户信息。

- 用户使用他们的凭证和模拟 header 进行 API 调用。
- API server 认证用户
- API server 确保经过身份验证的用户具有模拟权限。
- 请求用户的信息被替换为模拟值
- 请求被评估，授权作用于模拟的用户信息。

以下 HTTP header 可用户执行模拟请求：

- `Impersonate-User` ：充当的用户名
- `Impersonate-Group` ：作为组名。可以多次使用来设置多个组。可选的，需要 “Impersonate-User”
- `Impersonate-Extra-( extra name )` ：用于将额外字段与用户关联的动态 header。可选。需要 “Impersonate-User”

一组示例 header：

```
Impersonate-User: jane.doe@example.com
Impersonate-Group: developers
Impersonate-Group: admins
Impersonate-Extra-dn: cn=jane,ou=engineers,dc=example,dc=com
Impersonate-Extra-scopes: view
Impersonate-Extra-scopes: development
```

当使用 `kubectl` 的 `--as` 标志来配置 `Impersonate-User` header 时，可以使用 `--as-group` 标志来配置 `Impersonate-Group` header。

```
$ kubectl drain mynode
Error from server (Forbidden): User "clark" cannot get nodes at
the cluster scope. (get nodes mynode)

$ kubectl drain mynode --as=superman --as-group=system:masters
node "mynode" cordoned
node "mynode" draine
```

为模仿用户、组或设置额外字段，模拟用户必须能够对正在模拟的属性的种类（“用户”，“组”等）执行“模拟”动词。对于启用了 RBAC 授权插件的集群，以下 ClusterRole 包含设置用户和组模拟 header 所需的规则：

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: impersonator
rules:
- apiGroups: [""]
  resources: ["users", "groups", "serviceaccounts"]
  verbs: ["impersonate"]
```

额外的字段被评估为资源 “userextras” 的子资源。为了允许用户使用额外字段 “scope” 的模拟 header，应授予用户以下角色：

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: scopes-impersonator
# Can set "Impersonate-Extra-scopes" header.
- apiGroups: ["authentication.k8s.io"]
  resources: ["userextras/scopes"]
  verbs: ["impersonate"]
```

模拟 header 的可用值可以通过设置 `resourceNames` 可以使用的资源来限制。

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: limited-impersonator
rules:
# Can impersonate the user "jane.doe@example.com"
- apiGroups: [""]
  resources: ["users"]
  verbs: ["impersonate"]
  resourceNames: ["jane.doe@example.com"]

# Can impersonate the groups "developers" and "admins"
- apiGroups: [""]
  resources: ["groups"]
- verbs: ["impersonate"]
  resourceNames: ["developers","admins"]

# Can impersonate the extras field "scopes" with the values "vie
w" and "development"
- apiGroups: ["authentication.k8s.io"]
  resources: ["userextras/scopes"]
  verbs: ["impersonate"]
  resourceNames: ["view", "development"]
```

附录

创建证书

使用客户端证书进行身份验证时，可以使用现有的部署脚本或通过 `easyrsa` 或 `openssl` 手动生成证书。

使用已有的部署脚本

已有的部署脚本 在 `cluster/saltbase/salt/generate-cert/make-ca-cert.sh` 。

执行该脚本时需要传递两个参数。第一个参数是 API server 的 IP地址。第二个参数是 IP 形式的主题备用名称列表： `IP:<ip-address>` 或 `DNS:<dns-name>` 。

该脚本将生成三个文件： `ca.crt` 、 `server.crt` 和 `server.key` 。

最后，将一下参数添加到 API server 的启动参数中：

- `--client-ca-file=/srv/kubernetes/ca.crt`

- `--tls-cert-file=/srv/kubernetes/server.crt`
- `--tls-private-key-file=/srv/kubernetes/server.key`

easyrsa

easyrsa 可以用来手动为集群生成证书。

1. 下载，解压，并初始化修补版本的easyrsa3。

   ```
   curl -L -O https://storage.googleapis.com/kubernetes-release
   /easy-rsa/easy-rsa.tar.gz
   tar xzf easy-rsa.tar.gz
   cd easy-rsa-master/easyrsa3
   ./easyrsa init-pki
   ```

2. 生成 CA（使用 `--batch` 设置为自动模式。使用 `--req-cn` 设置默认的 CN）

   ```
   ./easyrsa --batch "--req-cn=${MASTER_IP}@`date +%s`" build-c
   a nopass
   ```

3. 生成服务器证书和密钥。（build-server-full [文件名]：生成一个键值对，在本地为客户端和服务器签名。）

   ```
   ./easyrsa --subject-alt-name="IP:${MASTER_IP}" build-server-
   full server nopass
   ```

4. 复制 `pki/ca.crt`，`pki/issued/server.crt` 和 `pki/private/server.key` 到您的目录下。

5. 将以下参数添加到 API server 的启动参数中：

   ```
   --client-ca-file=/yourdirectory/ca.crt
   --tls-cert-file=/yourdirectory/server.crt
   --tls-private-key-file=/yourdirectory/server.key
   ```

openssl

openssl 可以用来手动为集群生成证书。

1. 生成一个 2048 bit 的 ca.key：

```
openssl genrsa -out ca.key 2048
```

2. 根据 ca.key 生成一个 ca.crt（使用 -days 设置证书的有效时间）：

```
openssl req -x509 -new -nodes -key ca.key -subj "/CN=${MASTE
R_IP}" -days 10000 -out ca.crt
```

3. 生成一个 2048 bit 的 server.key：

```
openssl genrsa -out server.key 2048
```

4. 根据 server.key 生成一个 server.csr：

```
openssl req -new -key server.key -subj "/CN=${MASTER_IP}" -o
ut server.csr
```

5. 根据 ca.key、ca.crt 和 server.csr 生成 server.crt：

```
openssl x509 -req -in server.csr -CA ca.crt -CAkey ca.key -C
Acreateserial -out server.crt -days 10000
```

6. 查看证书：

```
openssl x509  -noout -text -in ./server.crt
```

最后，不要忘了向 API server 的启动参数中增加配置。

认证 API

您可以使用 `certificates.k8s.io` API将 x509 证书配置为用于身份验证，如 此处 所述。

官方v1.6文档地址：https://v1-6.docs.kubernetes.io/docs/admin/authentication/ 官方最新文档地址：https://kubernetes.io/docs/admin/authentication/

译者：Jimmy Song

访问 Kubernetes 集群

根据用户部署和暴露服务的方式不同，有很多种方式可以用来访问 kubernetes 集群。

- 最简单也是最直接的方式是使用 `kubectl` 命令。
- 其次可以使用 `kubeconfig` 文件来认证授权访问 API server。
- 通过各种 proxy 经过端口转发访问 kubernetes 集群中的服务
- 使用 Ingress，在集群外访问 kubernetes 集群内的 service

访问集群

第一次使用 kubectl 访问

如果您是第一次访问 Kubernetes API 的话，我们建议您使用 Kubernetes 命令行工具： `kubectl` 。

为了访问集群，您需要知道集群的地址，并且需要有访问它的凭证。通常，如果您完成了 入门指南 那么这些将会自动设置，或者其他人为您部署的集群提供并给您凭证和集群地址。

使用下面的命令检查 kubectl 已知的集群的地址和凭证：

```
$ kubectl config view
```

关于 kubectl 命令使用的更多 示例 和完整文档可以在这里找到：kubectl 手册

直接访问 REST API

Kubectl 处理对 apiserver 的定位和认证。如果您想直接访问 REST API，可以使用像 curl、wget 或浏览器这样的 http 客户端，有以下几种方式来定位和认证：

- 以 proxy 模式运行 kubectl。
  - 推荐方法。
  - 使用已保存的 apiserver 位置信息。
  - 使用自签名证书验证 apiserver 的身份。 没有 MITM（中间人攻击）的可能。
  - 认证到 apiserver。
  - 将来，可能会做智能的客户端负载均衡和故障转移。
- 直接向 http 客户端提供位置和凭据。
  - 替代方法。
  - 适用于通过使用代理而混淆的某些类型的客户端代码。
  - 需要将根证书导入浏览器以防止 MITM。

使用 kubectl proxy

以下命令作为反向代理的模式运行 kubectl。 它处理对 apiserver 的定位并进行认证。

像这样运行：

```
$ kubectl proxy --port=8080 &
```

查看关于 kubectl proxy 的更多细节。

然后您可以使用 curl、wget 或者浏览器来访问 API，如下所示：

```
$ curl http://localhost:8080/api/
{
  "versions": [
    "v1"
  ]
}
```

不使用 kubectl proxy（1.3.x 以前版本）

通过将认证 token 直接传递给 apiserver 的方式，可以避免使用 kubectl proxy，如下所示：

```
$ APISERVER=$(kubectl config view | grep server | cut -f 2- -d ":" | tr -d " ")
$ TOKEN=$(kubectl config view | grep token | cut -f 2 -d ":" | tr -d " ")
$ curl $APISERVER/api --header "Authorization: Bearer $TOKEN" --insecure
{
  "versions": [
    "v1"
  ]
}
```

不使用 kubectl proxy（1.3.x 以后版本）

在 Kubernetes 1.3 或更高版本中， `kubectl config view` 不再显示 token。 使用 `kubectl describe secret …` 获取 default service account 的 token，如下所示：

```
$ APISERVER=$(kubectl config view | grep server | cut -f 2- -d ":" | tr -d " ")
$ TOKEN=$(kubectl describe secret $(kubectl get secrets | grep default | cut -f1 -d ' ') | grep -E '^token' | cut -f2 -d':' | tr -d '\t')
$ curl $APISERVER/api --header "Authorization: Bearer $TOKEN" --insecure
{
  "kind": "APIVersions",
  "versions": [
    "v1"
  ],
  "serverAddressByClientCIDRs": [
    {
      "clientCIDR": "0.0.0.0/0",
      "serverAddress": "10.0.1.149:443"
    }
  ]
}
```

以上示例使用 `--insecure` 标志。 这使得它容易受到 MITM 攻击。 当 kubectl 访问集群时，它使用存储的根证书和客户端证书来访问服务器。 （这些安装在 `~/.kube` 目录中）。 由于集群证书通常是自签名的，因此可能需要特殊配置才能让您的 http 客户端使用根证书。

对于某些群集，apiserver 可能不需要身份验证；可以选择在本地主机上服务，或者使用防火墙保护。 对此还没有一个标准。配置对API的访问 描述了群集管理员如何配置此操作。 这种方法可能与未来的高可用性支持相冲突。

编程访问 API

Kubernetes 支持 Go 和 Python 客户端库。

Go 客户端

- 要获取该库，请运行以下命令： `go get k8s.io/client-go/<version number>/kubernetes` 请参阅 https://github.com/kubernetes/client-go 以查看支持哪些版本。
- 使用 client-go 客户端编程。请注意，client-go 定义了自己的 API 对象，因此如果需要，请从 client-go 而不是从主存储库导入 API 定义，例如导入 `k8s.io/client-go/1.4/pkg/api/v1` 是正确的。

Go 客户端可以使用与 kubectl 命令行工具相同的 kubeconfig 文件 来定位和验证 apiserver。参考官方 示例 和 client-go 示例。

如果应用程序在群集中以 Pod 的形式部署，请参考 下一节。

Python 客户端

要使用 Python client，请运行以下命令： `pip install kubernetes` 。查看 Python 客户端库页面 获取更多的安装选择。

Python 客户端可以使用与 kubectl 命令行工具相同的 kubeconfig 文件 来定位和验证 apiserver。参考该 示例。

其他语言

还有更多的 客户端库 可以用来访问 API。有关其他库的验证方式，请参阅文档。

在 Pod 中访问 API

在 Pod 中访问 API 时，定位和认证到 API server 的方式有所不同。在 Pod 中找到 apiserver 地址的推荐方法是使用kubernetes DNS 名称，将它解析为服务 IP，后者又将被路由到 apiserver。

向 apiserver 认证的推荐方法是使用 service account 凭据。通过 kube-system，pod 与 service account 相关联，并且将该 service account 的凭据（token）放入该 pod 中每个容器的文件系统树中，位于 `/var/run/secrets/kubernetes.io/serviceaccount/token` 。

如果可用，证书包将位于每个容器的文件系统树的 `/var/run/secrets/kubernetes.io/serviceaccount/ca.crt` 位置，并用于验证 apiserver 的服务证书。

最后，用于 namespace API 操作的默认 namespace 放在每个容器中的 `/var/run/secrets/kubernetes.io/serviceaccount/namespace` 中。

在 pod 中，连接到 API 的推荐方法是：

- 将 kubectl proxy 作为 pod 中的一个容器来运行，或作为在容器内运行的后台进程。它将 Kubernetes API 代理到 pod 的本地主机接口，以便其他任何 pod 中的容器内的进程都可以访问它。请参阅 在 pod 中使用 kubectl proxy 的示

例。

- 使用 Go 客户端库，并使用 `rest.InClusterConfig()` 和 `kubernetes.NewForConfig()` 函数创建一个客户端。

  他们处理对 apiserver 的定位和认证。示例

在以上的几种情况下，都需要使用 pod 的凭据与 apiserver 进行安全通信。

访问集群中运行的 service

上一节是关于连接到 kubernetes API server。这一节是关于连接到 kubernetes 集群中运行的 service。在 Kubernetes 中，node、 pod 和 services 都有它们自己的 IP。很多情况下，集群中 node 的 IP、Pod 的 IP、service 的 IP 都是不可路由的，因此在集群外面的机器就无法访问到它们，例如从您自己的笔记本电脑。

连接的方式

您可以选择以下几种方式从集群外部连接到 node、pod 和 service：

- 通过 public IP 访问 service。
  - 使用 `NodePort` 和 `LoadBalancer` 类型的 service，以使 service 能够在集群外部被访问到。请查看 service 和 kubectl expose 文档。
  - 根据您的群集环境，这可能会将服务暴露给您的公司网络，或者可能会将其暴露在互联网上。想想暴露的服务是否安全。它是否自己进行身份验证？
  - 将 pod 放在服务后面。 要从一组副本（例如为了调试）访问一个特定的 pod，请在 pod 上放置一个唯一的 label，并创建一个选择该 label 的新服务。
  - 在大多数情况下，应用程序开发人员不需要通过 node IP 直接访问节点。
- 通过 Proxy 规则访问 service、node、pod。
  - 在访问远程服务之前，请执行 apiserver 认证和授权。 如果服务不够安全，无法暴露给互联网，或者为了访问节点 IP 上的端口或进行调试，请使用这种方式。
  - 代理可能会导致某些 Web 应用程序出现问题。
  - 仅适用于 HTTP/HTTPS。
  - 见此描述。

- 在集群内访问 node 和 pod。
    - 运行一个 pod，然后使用 kubectl exec 命令连接到 shell。从该 shell 中连接到其他 node、pod 和 service。
    - 有些集群可能允许 ssh 到集群上的某个节点。 从那个节点您可以访问到集群中的服务。这是一个非标准的方法，它可能将在某些集群上奏效，而在某些集群不行。这些节点上可能安装了浏览器和其他工具也可能没有。群集 DNS 可能无法正常工作。

访问内置服务

通常集群内会有几个在 kube-system 中启动的服务。使用 `kubectl cluster-info` 命令获取该列表：

```
$ kubectl cluster-info

  Kubernetes master is running at https://104.197.5.247
  elasticsearch-logging is running at https://104.197.5.247/api/
v1/namespaces/kube-system/services/elasticsearch-logging/proxy
  kibana-logging is running at https://104.197.5.247/api/v1/name
spaces/kube-system/services/kibana-logging/proxy
  kube-dns is running at https://104.197.5.247/api/v1/namespaces
/kube-system/services/kube-dns/proxy
  grafana is running at https://104.197.5.247/api/v1/namespaces/
kube-system/services/monitoring-grafana/proxy
  heapster is running at https://104.197.5.247/api/v1/namespaces
/kube-system/services/monitoring-heapster/proxy
```

这显示了访问每个服务的代理 URL。

例如，此集群启用了集群级日志记录（使用Elasticsearch），如果传入合适的凭据，可以在该地址 `https://104.197.5.247/api/v1/namespaces/kube-system/services/elasticsearch-logging/proxy/` 访问到，或通过 kubectl 代理，例如：`http://localhost:8080/api/v1/namespaces/kube-system/services/elasticsearch-logging/proxy/`。

（有关如何传递凭据和使用 kubectl 代理，请 参阅上文）

手动构建 apiserver 代理 URL

如上所述，您可以使用 `kubectl cluster-info` 命令来检索服务的代理 URL。要创建包含服务端点、后缀和参数的代理 URL，您只需附加到服务的代理URL：

`http://` *kubernetes_master_address* `/api/v1/namespaces/` *namespace_name* `/services/` *service_name[:port_name]* `/proxy`

如果您没有指定 port 的名字，那么您不必在 URL 里指定 port_name。

示例

- 要想访问 Elasticsearch 的服务端点 `_search?q=user:kimchy`，您需要使用：`http://104.197.5.247/api/v1/namespaces/kube-system/services/elasticsearch-logging/proxy/_search?q=user:kimchy`
- 要想访问 Elasticsearch 的集群健康信息 `_cluster/health?pretty=true`，您需要使用：`https://104.197.5.247/api/v1/namespaces/kube-system/services/elasticsearch-logging/proxy/_cluster/health?pretty=true`

```json
{
  "cluster_name" : "kubernetes_logging",
  "status" : "yellow",
  "timed_out" : false,
  "number_of_nodes" : 1,
  "number_of_data_nodes" : 1,
  "active_primary_shards" : 5,
  "active_shards" : 5,
  "relocating_shards" : 0,
  "initializing_shards" : 0,
  "unassigned_shards" : 5
}
```

使用 web 浏览器来访问集群中运行的服务

您可以将 apiserver 代理网址放在浏览器的地址栏中。 然而：

- Web 浏览器通常不能传递 token，因此您可能需要使用基本（密码）认证。Apiserver 可以配置为接受基本认证，但您的集群可能未配置为接受基本认证。
- 某些网络应用程序可能无法正常工作，特别是那些在不知道代理路径前缀的情况下构造 URL 的客户端 JavaScript。

请求重定向

重定向功能已被弃用和删除。 请改用代理（见下文）。

多种代理

在使用 kubernetes 的时候您可能会遇到许多种不同的代理：

1. kubectl 代理：
    - 在用户桌面或 pod 中运行
    - 从 localhost 地址到 Kubernetes apiserver 的代理
    - 客户端到代理使用 HTTP
    - apiserver 的代理使用 HTTPS
    - 定位 apiserver
    - 添加身份验证 header
2. apiserver 代理：
    - 将一个堡垒机作为 apiserver
    - 将群集之外的用户连接到群集IP，否则可能无法访问
    - 在 apiserver 进程中运行
    - 客户端到代理使用 HTTPS（或 http，如果 apiserver 如此配置）
    - 根据代理目标的可用信息由代理选择使用 HTTP 或 HTTPS
    - 可用于访问 node、pod 或 service
    - 用于访问 service 时进行负载均衡
3. kube 代理：
    - 在每个节点上运行
    - 代理 UDP 和 TCP
    - 不支持 HTTP
    - 提供负载均衡
    - 只是用来访问 service
4. apiserver 前面的代理/负载均衡器：
    - 存在和实现因群集而异（例如 nginx）
    - 位于所有客户端和一个或多个 apiserver 之间
    - 作为负载均衡器，如果有多个 apiserver
5. 外部服务的云负载均衡器：
    - 由一些云提供商提供（例如 AWS ELB，Google Cloud Load Balancer）
    - 当 Kubernetes service 类型为 LoadBalancer 时，会自动创建
    - 仅使用 UDP/TCP
    - 实施方式因云提供商而异

使用 kubeconfig 文件配置跨集群认证

Kubernetes 的认证方式对于不同的人来说可能有所不同。

- 运行 kubelet 可能有一种认证方式（即证书）。
- 用户可能有不同的认证方式（即令牌）。
- 管理员可能具有他们为个人用户提供的证书列表。
- 我们可能有多个集群，并希望在同一个地方将其全部定义——这样用户就能使用自己的证书并重用相同的全局配置。

所以为了能够让用户轻松地在多个集群之间切换，对于多个用户的情况下，我们将其定义在了一个 kubeconfig 文件中。

此文件包含一系列与昵称相关联的身份验证机制和集群连接信息。它还引入了一个（用户）认证信息元组和一个被称为上下文的与昵称相关联的集群连接信息的概念。

如果明确指定，则允许使用多个 kubeconfig 文件。在运行时，它们与命令行中指定的覆盖选项一起加载并合并（参见下面的 规则）。

相关讨论

http://issue.k8s.io/1755

Kubeconfig 文件的组成

Kubeconifg 文件示例

```yaml
current-context: federal-context
apiVersion: v1
clusters:
- cluster:
    api-version: v1
    server: http://cow.org:8080
  name: cow-cluster
- cluster:
    certificate-authority: path/to/my/cafile
    server: https://horse.org:4443
  name: horse-cluster
- cluster:
    insecure-skip-tls-verify: true
    server: https://pig.org:443
  name: pig-cluster
contexts:
- context:
    cluster: horse-cluster
    namespace: chisel-ns
    user: green-user
  name: federal-context
- context:
    cluster: pig-cluster
    namespace: saw-ns
    user: black-user
  name: queen-anne-context
kind: Config
preferences:
  colors: true
users:
- name: blue-user
  user:
    token: blue-token
- name: green-user
  user:
    client-certificate: path/to/my/client/cert
    client-key: path/to/my/client/key
```

各个组件的拆解/释意

Cluster

```
clusters:
- cluster:
    certificate-authority: path/to/my/cafile
    server: https://horse.org:4443
  name: horse-cluster
- cluster:
    insecure-skip-tls-verify: true
    server: https://pig.org:443
  name: pig-cluster
```

`cluster` 中包含 kubernetes 集群的端点数据，包括 kubernetes apiserver 的完整 url 以及集群的证书颁发机构或者当集群的服务证书未被系统信任的证书颁发机构签名时，设置 `insecure-skip-tls-verify: true` 。

`cluster` 的名称（昵称）作为该 kubeconfig 文件中的集群字典的 key。 您可以使用 `kubectl config set-cluster` 添加或修改 `cluster` 条目。

user

```
users:
- name: blue-user
  user:
    token: blue-token
- name: green-user
  user:
    client-certificate: path/to/my/client/cert
    client-key: path/to/my/client/key
```

`user` 定义用于向 kubernetes 集群进行身份验证的客户端凭据。在加载/合并 kubeconfig 之后， `user` 将有一个名称（昵称）作为用户条目列表中的 key。 可用凭证有 `client-certificate` 、 `client-key` 、 `token` 和 `username/password` 。 `username/password` 和 `token` 是二者只能选择一个，但 client-certificate 和 client-key 可以分别与它们组合。

您可以使用 `kubectl config set-credentials` 添加或者修改 `user` 条目。

context

```
contexts:
- context:
    cluster: horse-cluster
    namespace: chisel-ns
    user: green-user
  name: federal-context
```

`context` 定义了一个命名的 `cluster` 、 `user` 、 `namespace` 元组，用于使用提供的认证信息和命名空间将请求发送到指定的集群。三个都是可选的；仅使用 `cluster` 、 `user` 、 `namespace` 之一指定上下文，或指定 none。未指定的值或在加载的 kubeconfig 中没有相应条目的命名值（例如，如果为上述 kubeconfig 文件指定了 `pink-user` 的上下文）将被替换为默认值。有关覆盖/合并行为，请参阅下面的 加载和合并规则。

您可以使用 `kubectl config set-context` 添加或修改上下文条目。

current-context

```
current-context: federal-context
```

`current-context` is the nickname or 'key' for the cluster,user,namespace tuple that kubectl will use by default when loading config from this file. You can override any of the values in kubectl from the commandline, by passing `--context=CONTEXT` , `--cluster=CLUSTER` , `--user=USER` , and/or `--namespace=NAMESPACE` respectively. You can change the `current-context` with `kubectl config use-context` .

`current-context` 是昵称或者说是作为 `cluster` 、 `user` 、 `namespace` 元组的 "key“，当 kubectl 从该文件中加载配置的时候会被默认使用。您可以在 kubectl 命令行里覆盖这些值，通过分别传入 `—context=CONTEXT` 、 `—cluster=CLUSTER` 、 `--user=USER` 和 `--namespace=NAMESPACE` 。

您可以使用 `kubectl config use-context` 更改 `current-context` 。

```
apiVersion: v1
kind: Config
preferences:
  colors: true
```

杂项

`apiVersion` 和 `kind` 标识客户端解析器的版本和模式，不应手动编辑。 `preferences` 指定可选（和当前未使用）的 kubectl 首选项。

查看 kubeconfig 文件

`kubectl config view` 命令可以展示当前的 kubeconfig 设置。默认将为您展示所有的 kubeconfig 设置；您可以通过传入 `—minify` 参数，将视图过滤到与 `current-context` 有关的配额设置。有关其他选项，请参阅 `kubectl config view` 。

构建您自己的 kubeconfig 文件

您可以使用上文 示例 kubeconfig 文件 作为

注意： 如果您是通过 `kube-up.sh` 脚本部署的 kubernetes 集群，不需要自己创建 kubeconfig 文件——该脚本已经为您创建过了。

{:.note}

当 api server 启动的时候使用了 `—token-auth-file=tokens.csv` 选项时，上述文件将会与 API server 相关联， `tokens.csv` 文件看起来会像这个样子：

```
blue-user,blue-user,1
mister-red,mister-red,2
```

注意： 启动 API server 时有很多 可用选项。请您一定要确保理解您使用的选项。

上述示例 kubeconfig 文件提供了 `green-user` 的客户端凭证。因为用户的 `current-user` 是 `green-user` ，任何该 API server 的客户端使用该示例 kubeconfig 文件时都可以成功登录。同样，我们可以通过修改 `current-context` 的值以 `blue-user` 的身份操作。

在上面的示例中， `green-user` 通过提供凭据登录， `blue-user` 使用的是 token。使用 `kubectl config set-credentials` 指定登录信息。想了解更多信息，请访问 "示例文件相关操作命令"。

加载和合并规则

加载和合并 kubeconfig 文件的规则很简单，但有很多。最终的配置按照以下顺序构建：

1. 从磁盘中获取 kubeconfig。这将通过以下层次结构和合并规则完成：

   如果设置了 `CommandLineLocation` （ `kubeconfig` 命令行参数的值），将会只使用该文件，而不会进行合并。该参数在一条命令中只允许指定一次。

   或者，如果设置了 `EnvVarLocation` （ `$KUBECONFIG` 的值），其将会被作为应合并的文件列表，并根据以下规则合并文件。空文件名被忽略。非串行内容的文件将产生错误。设置特定值或 map key 的第一个文件将优先使用，并且值或 map key 也永远不会更改。 这意味着设置 CurrentContext 的第一个文件将保留其上下文。 这也意味着如果两个文件同时指定一个 `red-user` ，那么将只使用第一个文件中的 `red-user` 的值。 即使第二个文件的 `red-user` 中有非冲突条目也被丢弃。

   另外，使用 Home 目录位置（ `~/.kube/config` ）将不会合并。

2. 根据此链中的第一个命中确定要使用的上下文

   i. 命令行参数—— `context` 命令行选项的值
   ii. 来自合并后的 `kubeconfig` 文件的 `current-context`
   iii. 在这个阶段允许空

3. 确定要使用的群集信息和用户 。此时，我们可能有也可能没有上下文。他们是基于这个链中的第一次命中。 （运行两次，一次为用户，一次为集群）

   i. 命令行参数—— `user` 指定用户， `cluster` 指定集群名称
   ii. 如果上下文存在，则使用上下文的值
   iii. 允许空

4. 确定要使用的实际群集信息。此时，我们可能有也可能没有集群信息。根据链条构建每个集群信息（第一次命中胜出）：

   i. 命令行参数—— `server` ， `api-version` ， `certificate-authority` 和 `insecure-skip-tls-verify`
   ii. 如果存在集群信息，并且存在该属性的值，请使用它。
   iii. 如果没有服务器位置，则产生错误。

5. 确定要使用的实际用户信息。用户使用与集群信息相同的规则构建，除非，您的每个用户只能使用一种认证技术。

   i. 负载优先级为1）命令行标志 2）来自 kubeconfig 的用户字段
   ii. 命令行标志是：`client-certificate` 、 `client-key` 、 `username` 、 `password` 和 `token`
   iii. 如果有两种冲突的技术，则失败。
6. 对于任何仍然缺少的信息，将使用默认值，并可能会提示验证信息
7. Kubeconfig 文件中的所有文件引用都相对于 kubeconfig 文件本身的位置进行解析。当命令行上显示文件引用时，它们将相对于当前工作目录进行解析。当路径保存在 `~/.kube/config` 中时，相对路径使用相对存储，绝对路径使用绝对存储。

Kubeconfig 文件中的任何路径都相对于 kubeconfig 文件本身的位置进行解析。

使用 `kubectl config <subcommand>` 操作 kubeconfig

`kubectl config` 有一些列的子命令可以帮助我们更方便的操作 kubeconfig 文件。

请参阅 kubectl/kubectl_config。

Example

```
$ kubectl config set-credentials myself --username=admin --password=secret
$ kubectl config set-cluster local-server --server=http://localhost:8080
$ kubectl config set-context default-context --cluster=local-server --user=myself
$ kubectl config use-context default-context
$ kubectl config set contexts.default-context.namespace the-right-prefix
$ kubectl config view
```

产生如下输出：

```yaml
apiVersion: v1
clusters:
- cluster:
    server: http://localhost:8080
  name: local-server
contexts:
- context:
    cluster: local-server
    namespace: the-right-prefix
    user: myself
  name: default-context
current-context: default-context
kind: Config
preferences: {}
users:
- name: myself
  user:
    password: secret
    username: admin
```

Kubeconfig 文件会像这样子：

```yaml
apiVersion: v1
clusters:
- cluster:
    server: http://localhost:8080
  name: local-server
contexts:
- context:
    cluster: local-server
    namespace: the-right-prefix
    user: myself
  name: default-context
current-context: default-context
kind: Config
preferences: {}
users:
- name: myself
  user:
    password: secret
    username: admin
```

示例文件相关操作命令

```
$ kubectl config set preferences.colors true
$ kubectl config set-cluster cow-cluster --server=http://cow.org:8080 --api-version=v1
$ kubectl config set-cluster horse-cluster --server=https://horse.org:4443 --certificate-authority=path/to/my/cafile
$ kubectl config set-cluster pig-cluster --server=https://pig.org:443 --insecure-skip-tls-verify=true
$ kubectl config set-credentials blue-user --token=blue-token
$ kubectl config set-credentials green-user --client-certificate=path/to/my/client/cert --client-key=path/to/my/client/key
$ kubectl config set-context queen-anne-context --cluster=pig-cluster --user=black-user --namespace=saw-ns
$ kubectl config set-context federal-context --cluster=horse-cluster --user=green-user --namespace=chisel-ns
$ kubectl config use-context federal-context
```

最后将它们捆绑在一起

所以，将这一切绑在一起，快速创建自己的 kubeconfig 文件：

- 仔细看一下，了解您的 api-server 的启动方式：在设计 kubeconfig 文件以方便身份验证之前，您需要知道您自己的安全要求和策略。
- 将上面的代码段替换为您的集群的 api-server 端点的信息。
- 确保您的 api-server 至少能够以提供一个用户（即 `green-user` ）凭据的方式启动。 当然您必须查看 api-server 文档，以了解当前关于身份验证细节方面的最新技术。

通过端口转发访问集群中的应用程序

本页向您展示如何使用 `kubectl port-forward` 命令连接到运行在 Kubernetes 集群中的 Redis 服务器。这种类型的连接对于数据库调试很有帮助。

创建一个 Pod 来运行 Redis 服务器

1. 创建一个 Pod：

   ```
   kubectl create -f https://k8s.io/docs/tasks/access-applicati
   on-cluster/redis-master.yaml
   ```

   命令运行成功后将有以下输出验证该 Pod 是否已经创建：

   ```
   pod "redis-master" created
   ```

2. 检查 Pod 是否正在运行且处于就绪状态：

   ```
   kubectl get pods
   ```

   当 Pod 就绪，输出显示 Running 的状态：

   ```
    NAME              READY     STATUS    RESTARTS   AGE
    redis-master      2/2       Running   0          41s
   ```

3. 验证 Redis 服务器是否已在 Pod 中运行，并监听 6379 端口：

   ```
   kubectl get pods redis-master --template='{{(index (index .s
   pec.containers 0).ports 0).containerPort}}{{"\n"}}'
   ```

   端口输出如下：

   ```
   6379
   ```

将本地端口转发到 Pod 中的端口

1. 将本地工作站上的 6379 端口转发到 redis-master pod 的 6379 端口：

   ```
   kubectl port-forward redis-master 6379:6379
   ```

   输出类似于：

   ```
    I0710 14:43:38.274550    3655 portforward.go:225] Forwardin
   g from 127.0.0.1:6379 -> 6379
    I0710 14:43:38.274797    3655 portforward.go:225] Forwardin
   g from [::1]:6379 -> 6379
   ```

2. 启动 Redis 命令行界面

   ```
   redis-cli
   ```

3. 在 Redis 命令行提示符下，输入 `ping` 命令：

   ```
   127.0.0.1:6379>ping
   ```

   Ping 请求成功返回 PONG。

讨论

创建连接，将本地的 6379 端口转发到运行在 Pod 中的 Redis 服务器的 6379 端口。有了这个连接您就可以在本地工作站中调试运行在 Pod 中的数据库。

使用 service 访问群集中的应用程序

本文向您展示如何创建 Kubernetes Service 对象，外部客户端可以使用它来访问集群中运行的应用程序。该 Service 可以为具有两个运行实例的应用程序提供负载均衡。

目的

- 运行 Hello World 应用程序的两个实例。
- 创建一个暴露 node 节点端口的 Service 对象。
- 使用 Service 对象访问正在运行的应用程序。

为在两个 pod 中运行的应用程序创建 service

1. 在集群中运行 Hello World 应用程序：

```
kubectl run hello-world --replicas=2 --labels="run=load-balancer-example" --image=gcr.io/google-samples/node-hello:1.0 --port=8080
```

上述命令创建一个 Deployment 对象和一个相关联的 ReplicaSet 对象。该 ReplicaSet 有两个 Pod，每个 Pod 中都运行一个 Hello World 应用程序。

2. 显示关于该 Deployment 的信息：

```
kubectl get deployments hello-world
kubectl describe deployments hello-world
```

3. 显示 ReplicaSet 的信息：

```
kubectl get replicasets
kubectl describe replicasets
```

4. 创建一个暴露该 Deployment 的 Service 对象：

```
kubectl expose deployment hello-world --type=NodePort --name
=example-service
```

5. 显示该 Service 的信息：

```
kubectl describe services example-service
```

输出类似于：

```
Name:                   example-service
Namespace:              default
Labels:                 run=load-balancer-example
Selector:               run=load-balancer-example
Type:                   NodePort
IP:                     10.32.0.16
Port:                   <unset> 8080/TCP
NodePort:               <unset> 31496/TCP
Endpoints:              10.200.1.4:8080,10.200.2.5:8080
Session Affinity:       None
No events.
```

记下服务的 NodePort 值。例如，在前面的输出中，NodePort 值为 31496。

6. 列出运行 Hello World 应用程序的 Pod：

```
kubectl get pods --selector="run=load-balancer-example" --ou
tput=wide
```

输出类似于：

```
NAME                          READY   STATUS    ...  IP
       NODE
hello-world-2895499144-bsbk5  1/1     Running   ...  10.20
0.1.4    worker1
hello-world-2895499144-m1pwt  1/1     Running   ...  10.20
0.2.5    worker2
```

7. 获取正在运行 Hello World 应用程序的 Pod 的其中一个节点的 public IP 地址。如何得到这个地址取决于您的集群设置。例如，如果您使用 Minikube，可以通过运行 `kubectl cluster-info` 查看节点地址。如果您是使用 Google

Compute Engine 实例，可以使用 `gcloud compute instances list` 命令查看您的公共地址节点。

8. 在您选择的节点上，在您的节点端口上例如创建允许 TCP 流量的防火墙规则，如果您的服务 NodePort 值为 31568，创建防火墙规则，允许端口 31568 上的TCP流量。
9. 使用节点地址和节点端口访问 Hello World 应用程序：

    ```
    curl http://<public-node-ip>:<node-port>
    ```

    其中 `<public-node-ip>` 是您节点的 public IP地址，而 `<node-port>` 是您服务的 NodePort 值。

    对成功请求的响应是一个 hello 消息：

    ```
    Hello Kubernetes!
    ```

使用 Service 配置文件

作为使用 `kubectl expose` 的替代方法，您可以使用 service 配置文件 来创建 Service。

要删除 Service，输入以下命令：

```
kubectl delete services example-service
```

删除 Deployment、ReplicaSet 和正运行在 Pod 中的 Hello World 应用程序，输入以下命令：

```
kubectl delete deployment hello-world
```

了解更多关于 使用 service 连接应用程序。

从外部访问Kubernetes中的Pod

前面几节讲到如何访问kubneretes集群，本文主要讲解访问kubenretes中的Pod和Serivce的集中方式，包括如下几种：

- hostNetwork
- hostPort
- NodePort
- LoadBalancer
- Ingress

说是暴露Pod其实跟暴露Service是一回事，因为Pod就是Service的backend。

hostNetwork: true

这是一种直接定义Pod网络的方式。

如果在Pod中使用 `hostNotwork:true` 配置的话，在这种pod中运行的应用程序可以直接看到pod启动的主机的网络接口。在主机的所有网络接口上都可以访问到该应用程序。以下是使用主机网络的pod的示例定义：

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: influxdb
spec:
  hostNetwork: true
  containers:
    - name: influxdb
      image: influxdb
```

部署该Pod：

```
$ kubectl create -f influxdb-hostnetwork.yml
```

访问该pod所在主机的8086端口：

```
curl -v http://$POD_IP:8086/ping
```

将看到204 No Content的204返回码，说明可以正常访问。

注意每次启动这个Pod的时候都可能被调度到不同的节点上，所有外部访问Pod的IP也是变化的，而且调度Pod的时候还需要考虑是否与宿主机上的端口冲突，因此一般情况下除非您知道需要某个特定应用占用特定宿主机上的特定端口时才使用 `hostNetwork: true` 的方式。

这种Pod的网络模式有一个用处就是可以将网络插件包装在Pod中然后部署在每个宿主机上，这样该Pod就可以控制该宿主机上的所有网络。

hostPort

这是一种直接定义Pod网络的方式。

`hostPort` 是直接将容器的端口与所调度的节点上的端口路由，这样用户就可以通过宿主机的IP加上来访问Pod了，如:。

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: influxdb
spec:
  containers:
    - name: influxdb
      image: influxdb
      ports:
        - containerPort: 8086
          hostPort: 8086
```

这样做有个缺点，因为Pod重新调度的时候该Pod被调度到的宿主机可能会变动，这样就变化了，用户必须自己维护一个Pod与所在宿主机的对应关系。

这种网络方式可以用来做 nginx Ingress controller。外部流量都需要通过kubenretes node节点的80和443端口。

NodePort

NodePort在kubenretes里是一个广泛应用的服务暴露方式。Kubernetes中的service默认情况下都是使用的 `ClusterIP` 这种类型，这样的service会产生一个ClusterIP，这个IP只能在集群内部访问，要想让外部能够直接访问service，需要将service type修改为 `nodePort` 。

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: influxdb
  labels:
    name: influxdb
spec:
  containers:
    - name: influxdb
      image: influxdb
      ports:
        - containerPort: 8086
```

同时还可以给service指定一个 `nodePort` 值，范围是30000-32767，这个值在API server的配置文件中，用 `--service-node-port-range` 定义。

```yaml
kind: Service
apiVersion: v1
metadata:
  name: influxdb
spec:
  type: NodePort
  ports:
    - port: 8086
      nodePort: 30000
  selector:
    name: influxdb
```

集群外就可以使用kubernetes任意一个节点的IP加上30000端口访问该服务了。kube-proxy会自动将流量以round-robin的方式转发给该service的每一个pod。

这种服务暴露方式，无法让你指定自己想要的应用常用端口，不过可以在集群上再部署一个反向代理作为流量入口。

LoadBalancer

`LoadBalancer` 只能在service上定义。这是公有云提供的负载均衡器，如AWS、Azure、CloudStack、GCE等。

```yaml
kind: Service
apiVersion: v1
metadata:
  name: influxdb
spec:
  type: LoadBalancer
  ports:
    - port: 8086
  selector:
    name: influxdb
```

查看服务：

```
$ kubectl get svc influxdb
NAME        CLUSTER-IP      EXTERNAL-IP     PORT(S)          AGE
influxdb    10.97.121.42    10.13.242.236   8086:30051/TCP   39s
```

内部可以使用ClusterIP加端口来访问服务，如19.97.121.42:8086。

外部可以用以下两种方式访问该服务：

- 使用任一节点的IP加30051端口访问该服务
- 使用 `EXTERNAL-IP` 来访问，这是一个VIP，是云供应商提供的负载均衡器IP，如10.13.242.236:8086。

Ingress

`Ingress` 是自kubernetes1.1版本后引入的资源类型。必须要部署Ingress controller才能创建Ingress资源，Ingress controller是以一种插件的形式提供。Ingress controller 是部署在Kubernetes之上的Docker容器。它的Docker镜像包含一个像nginx或HAProxy的负载均衡器和一个控制器守护进程。控制器守护程序从Kubernetes接收所需的Ingress配置。它会生成一个nginx或HAProxy配置文件，并重新启动负载平衡器进程以使更改生效。换句话说，Ingress controller是由Kubernetes管理的负载均衡器。

Kubernetes Ingress提供了负载平衡器的典型特性：HTTP路由，粘性会话，SSL终止，SSL直通，TCP和UDP负载平衡等。目前并不是所有的Ingress controller都实现了这些功能，需要查看具体的Ingress controller文档。

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: influxdb
spec:
  rules:
    - host: influxdb.kube.example.com
      http:
        paths:
          - backend:
              serviceName: influxdb
              servicePort: 8086
```

外部访问URL http://influxdb.kube.example.com/ping 访问该服务，入口就是80端口，然后Ingress controller直接将流量转发给后端Pod，不需再经过kube-proxy的转发，比LoadBalancer方式更高效。

总结

总的来说Ingress是一个非常灵活和越来越得到厂商支持的服务暴露方式，包括Nginx、HAProxy、Traefik，还有各种Service Mesh，而其它服务暴露方式可以更适用于服务调试、特殊应用的部署。

参考

- Accessing Kubernetes Pods from Outside of the Cluster - alesnosek.com

Cabin - Kubernetes手机客户端

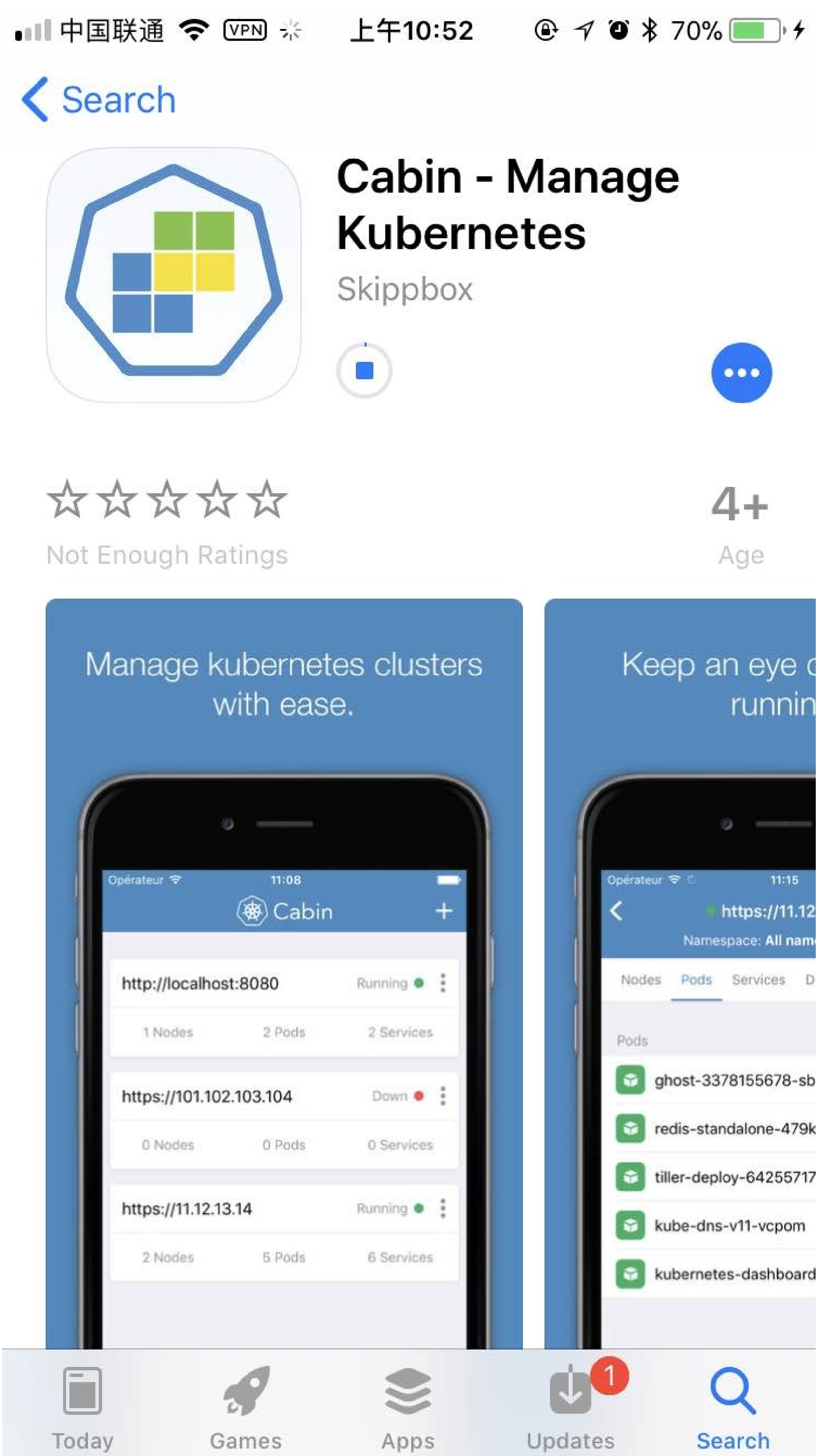
cabin是由bitnami开源的手机管理Kubernetes集群的客户端，目前提供iOS和安卓版本，代码开源在GitHub上：https://bitnami.com/

为了方便移动办公，可以使用Cabin这个kuberntes手机客户端，可以链接GKE和任何Kubernetes集群，可以使用以下三种认证方式：

- 证书
- token
- kubeconfig文件

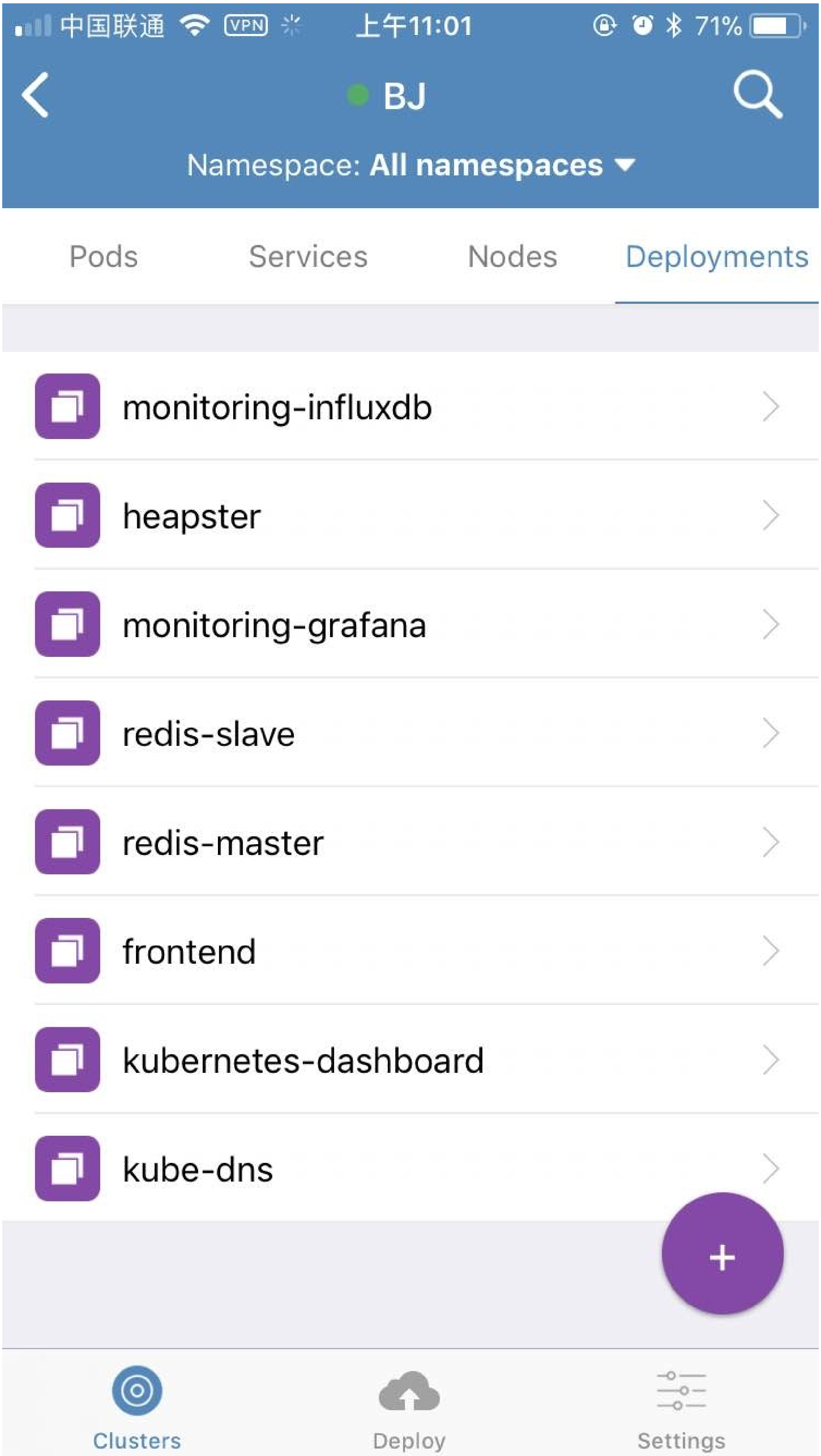
所有功能跟kubernetes dashboard相同，还可以支持使用Helm chart部署应用，可以配置自定义的chart仓库地址。

iPhone用户可以在App Store中搜索**Cabin**即可找到。


中国联通
上午10:52
70%
Search
Cabin - Manage Kubernetes
Skippbox
Not Enough Ratings
4+
Age
Manage kubernetes clusters with ease.
Cabin
http://localhost:8080
Running
1 Nodes
2 Pods
2 Services
https://101.102.103.104
Down
0 Nodes
0 Pods
0 Services
https://11.12.13.14
Running
2 Nodes
5 Pods
6 Services
Nodes
Pods
Services
ghost-3378155678-sb
redis-standalone-479k
tiller-deploy-64255717
kube-dns-v11-vcpom
kubernetes-dashboard
Today
Games
Apps
Updates
Search

图片 - *App Store*

可以很方便的在手机上操作自己的kubernetes集群，还可以登录到容器中操作，只要是kubernetes API支持的功能，都可以在该移动客户端上实现。


中国联通 VPN 上午11:01 71%
BJ
Namespace: All namespaces
Pods
Services
Nodes
Deployments
monitoring-influxdb
heapster
monitoring-grafana
redis-slave
redis-master
frontend
kubernetes-dashboard
kube-dns
+
Clusters
Deploy
Settings

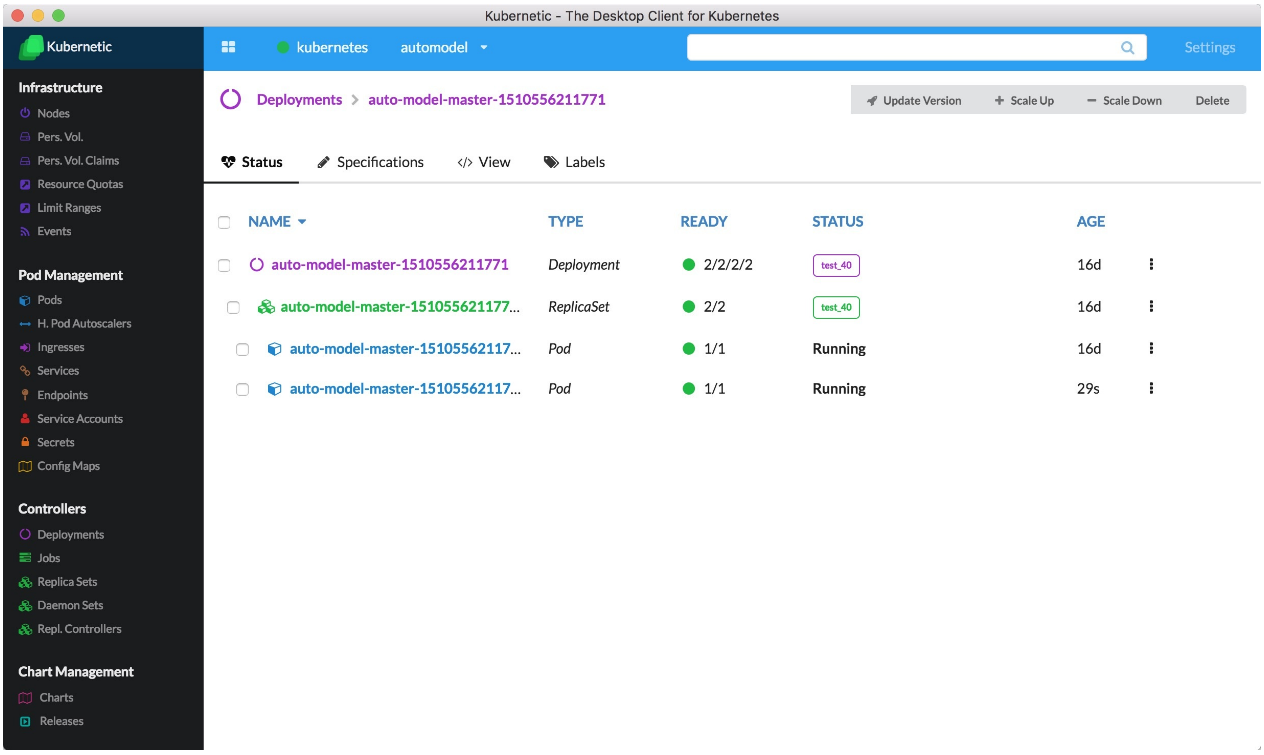
图片 - 在手机上操作*Kubernetes*集群

更多详细信息请参考：https://github.com/bitnami/cabin

Kubernetic - Kubernetes桌面客户端

Kubernetic - 一款kubenretes桌面客户端，https://kubernetic.com/，支持以下特性：

- 实时展示集群状态
- 多集群，多个namespace管理
- 原生kubernetes支持
- 支持使用chart安装应用
- 使用kubeconfig登陆认证



图片 - *Kubernetic客户端*

该客户端支持Mac和Windows系统，beta版本免费使用，stable版本需要付费。beta版本某些功能不完善，比如无法在应用内修改ingress配置，无法监控应用的状态等。

Kubernator - 更底层的Kubernetes UI

Kubernator相较于Kubernetes Dashboard来说，是一个更底层的Kubernetes UI，Dashboard操作的都是Kubernetes的底层对象，而Kubernator是直接操作Kubernetes各个对象的YAML文件。

Kubernator提供了一种基于目录树和关系拓扑图的方式来管理Kubernetes的对象的方法，用户可以在Web上像通过GitHub的网页版一样操作Kubernetes的对象，执行修改、拷贝等操作，详细的使用方式见https://github.com/smpio/kubernator 。

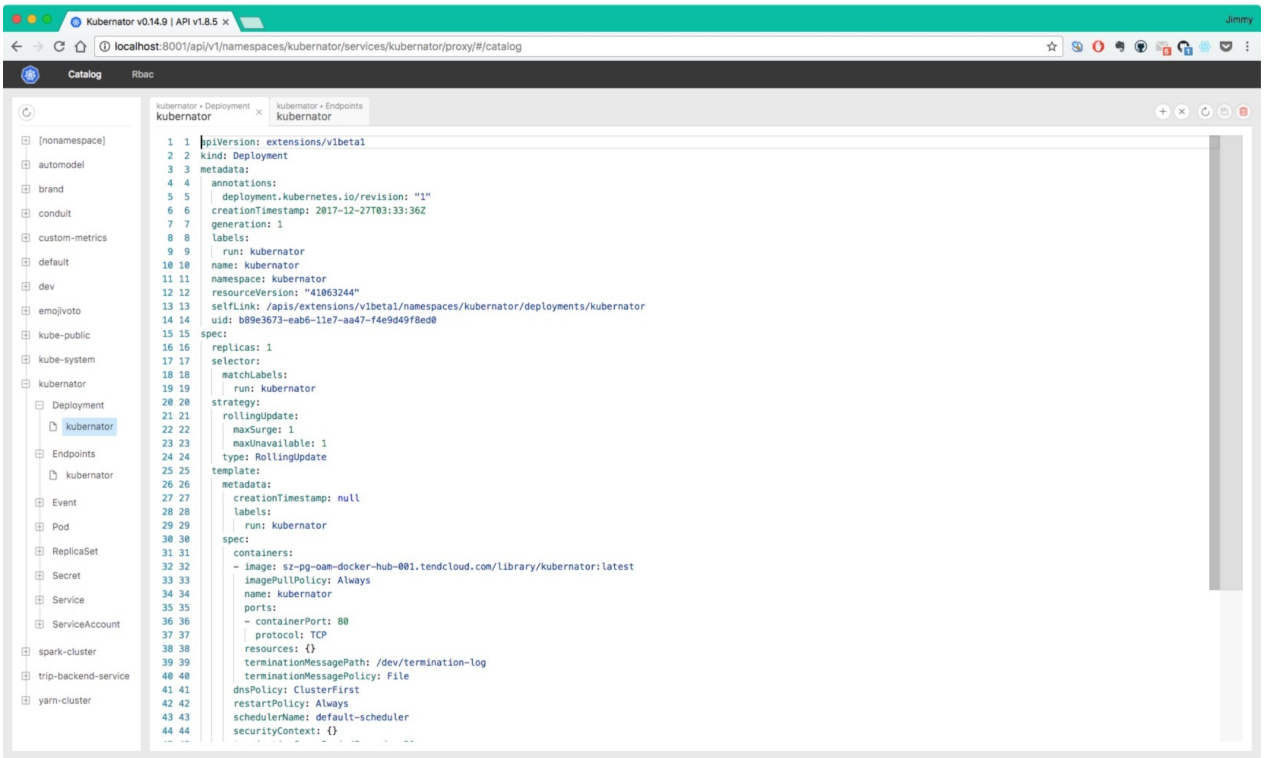
安装Kubernator

Kubernator的安装十分简单，可以直接使用 `kubectl` 命令来运行，它不依赖任何其它组件。

```
kubectl create ns kubernator
kubectl -n kubernator run --image=smpio/kubernator --port=80 kub
ernator
kubectl -n kubernator expose deploy kubernator
kubectl proxy
```

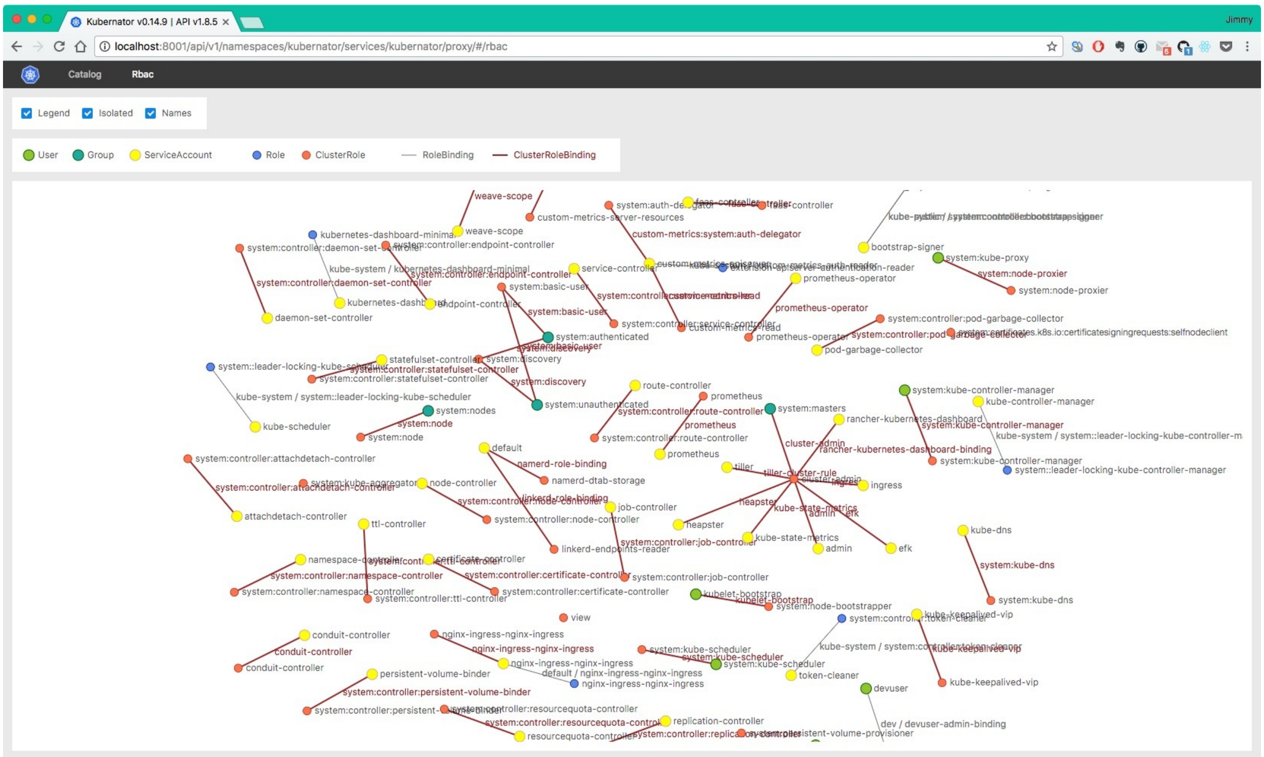
然后就可以通过
http://localhost:8001/api/v1/namespaces/kubernator/services/kubernator/proxy/来访问了。

Catalog页面可以看到Kubernetes中资源对象的树形结构，还可以在该页面中对资源对象的配置进行更改和操作。



图片 - *Kubernator catalog*页面

Rbac页面可以看到集群中RBAC关系及结构。



图片 - Kubernator rbac页面

在 Kubernetes 中开发部署应用

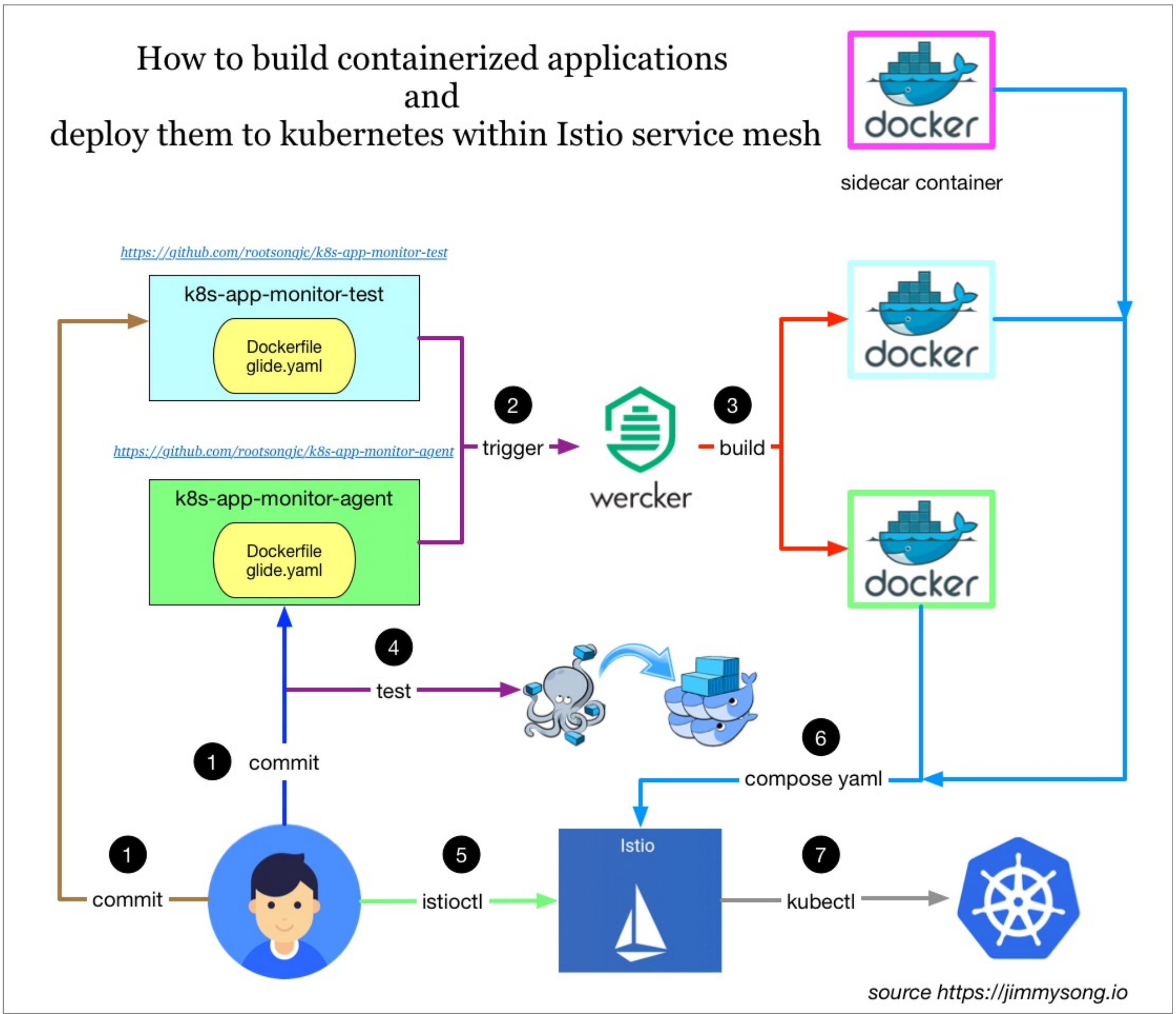
理论上只要可以使用主机名做服务注册的应用都可以迁移到 kubernetes 集群上。看到这里你可能不禁要问，为什么使用 IP 地址做服务注册发现的应用不适合迁移到 kubernetes 集群？因为这样的应用不适合自动故障恢复，因为目前 kubernetes 中不支持固定 Pod 的 IP 地址，当 Pod 故障后自动转移到其他 Node 的时候该 Pod 的 IP 地址也随之变化。

将传统应用迁移到 kubernetes 中可能还有很长的路要走，但是直接开发 Cloud native 应用，kubernetes 就是最佳运行时环境了。

适用于kubernetes的应用开发部署流程

本文讲解了如何开发容器化应用，并使用Wercker持续集成工具构建docker镜像上传到docker镜像仓库中，然后在本地使用*docker-compose*测试后，再使用 `kompose` 自动生成kubernetes的yaml文件，再将注入Envoy sidecar容器，集成Istio service mesh中的详细过程。

整个过程如下图所示。

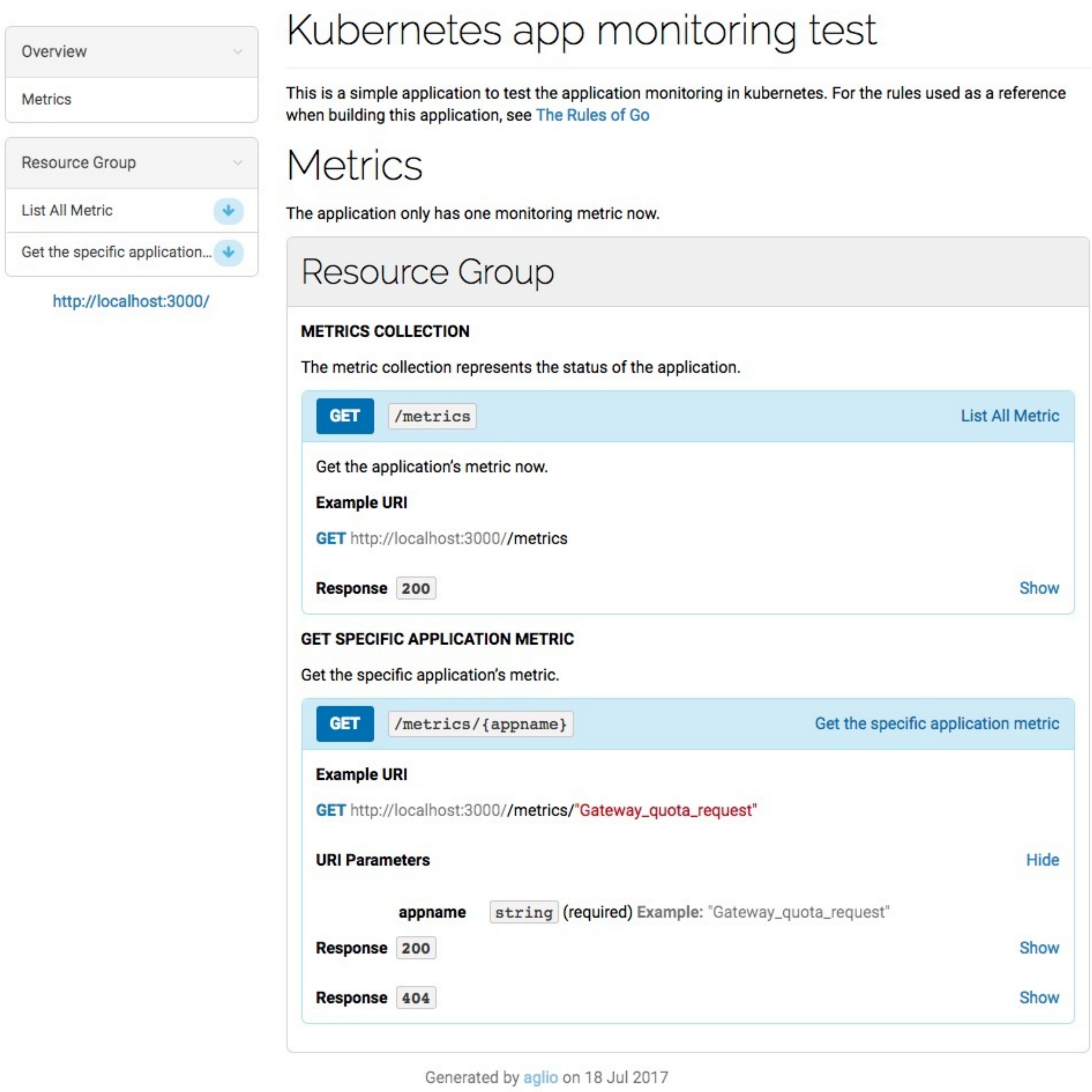


图片 - 流程图

为了讲解详细流程，我特意写了用Go语言开发的示例程序放在GitHub中，模拟监控流程：

- k8s-app-monitor-test：生成模拟的监控数据，在接收到http请求返回json格式的metrics信息
- K8s-app-monitor-agent：获取监控metrics信息并绘图，访问该服务将获得监控图表

API文档见k8s-app-monitor-test中的 `api.html` 文件，该文档在API blueprint中定义，使用aglio生成，打开后如图所示：



图片 - *API*

关于服务发现

`K8s-app-monitor-agent` 服务需要访问 `k8s-app-monitor-test` 服务，这就涉及到服务发现的问题，我们在代码中直接写死了要访问的服务的内网DNS地址（kubedns中的地址，即 `k8s-app-monitor-test.default.svc.cluster.local` ）。

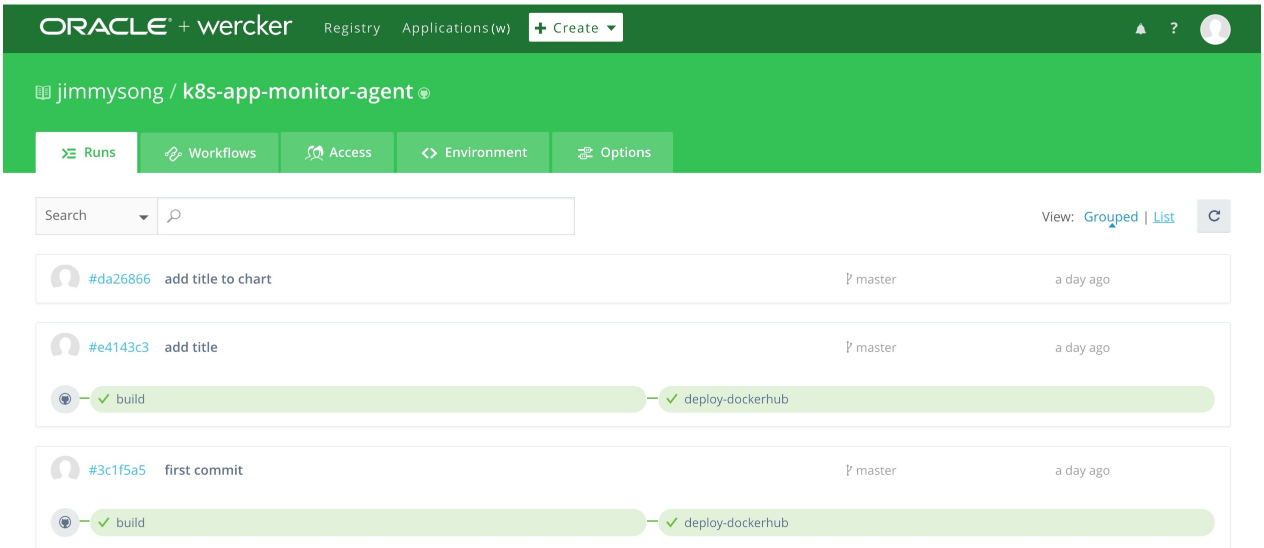
我们知道Kubernetes在启动Pod的时候为容器注入环境变量，这些环境变量在所有的 namespace 中共享（环境变量是不断追加的，新启动的Pod中将拥有老的Pod中所有的环境变量，而老的Pod中的环境变量不变）。但是既然使用这些环境变量就已经可以访问到对应的service，那么获取应用的地址信息，究竟是使用变量呢？还是直接使用DNS解析来发现？

答案是使用DNS，详细说明见Kubernetes中的服务发现与Docker容器间的环境变量传递源码探究。

持续集成

因为我使用wercker自动构建，构建完成后自动打包成docker镜像并上传到docker hub中（需要现在docker hub中创建repo）。

构建流程见：https://app.wercker.com/jimmysong/k8s-app-monitor-agent/



图片 - *wercker构建页面*

生成了如下两个docker镜像：

- jimmysong/k8s-app-monitor-test:9c935dd
- jimmysong/k8s-app-monitor-agent:234d51c

测试

在将服务发布到线上之前，我们可以先使用*docker-compose*在本地测试一下，这两个应用的 `docker-compose.yaml` 文件如下：

```yaml
version: '2'
services:
  k8s-app-monitor-agent:
    image: jimmysong/k8s-app-monitor-agent:234d51c
    container_name: monitor-agent
    depends_on:
      - k8s-app-monitor-test
    ports:
      - 8888:8888
    environment:
      - SERVICE_NAME=k8s-app-monitor-test
  k8s-app-monitor-test:
    image: jimmysong/k8s-app-monitor-test:9c935dd
    container_name: monitor-test
    ports:
      - 3000:3000
```

执行下面的命令运行测试。

```
docker-compose up
```

在浏览器中访问http://localhost:8888/k8s-app-monitor-test就可以看到监控页面。

发布

所有的kubernetes应用启动所用的yaml配置文件都保存在那两个GitHub仓库的 `manifest.yaml` 文件中。也可以使用kompose这个工具，可以将*docker-compose*的YAML文件转换成kubernetes规格的YAML文件。

分别在两个GitHub目录下执行 `kubectl create -f manifest.yaml` 即可启动服务。也可以直接在*k8s-app-monitor-agent*代码库的 `k8s` 目录下执行 `kubectl apply -f kompose` 。

在以上YAML文件中有包含了Ingress配置，是为了将*k8s-app-monitor-agent*服务暴露给集群外部访问。

方式一

服务启动后需要更新ingress配置，在ingress.yaml文件中增加以下几行：

```
  - host: k8s-app-monitor-agent.jimmysong.io
    http:
      paths:
      - path: /k8s-app-monitor-agent
        backend:
          serviceName: k8s-app-monitor-agent
          servicePort: 8888
```

保存后，然后执行 `kubectl replace -f ingress.yaml` 即可刷新ingress。

修改本机的 `/etc/hosts` 文件，在其中加入以下一行：

```
172.20.0.119 k8s-app-monitor-agent.jimmysong.io
```

当然你也可以将该域名加入到内网的DNS中，为了简单起见我使用hosts。

方式二

或者不修改已有的Ingress，而是为该队外暴露的服务单独创建一个Ingress，如下：

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: k8s-app-monitor-agent-ingress
  annotations:
    kubernetes.io/ingress.class: "treafik"
spec:
  rules:
  - host: k8s-app-monitor-agent.jimmysong.io
    http:
      paths:
      - path: /
        backend:
          serviceName: k8s-app-monitor-agent
          servicePort: 8888
```

详见边缘节点配置。

集成Istio service mesh

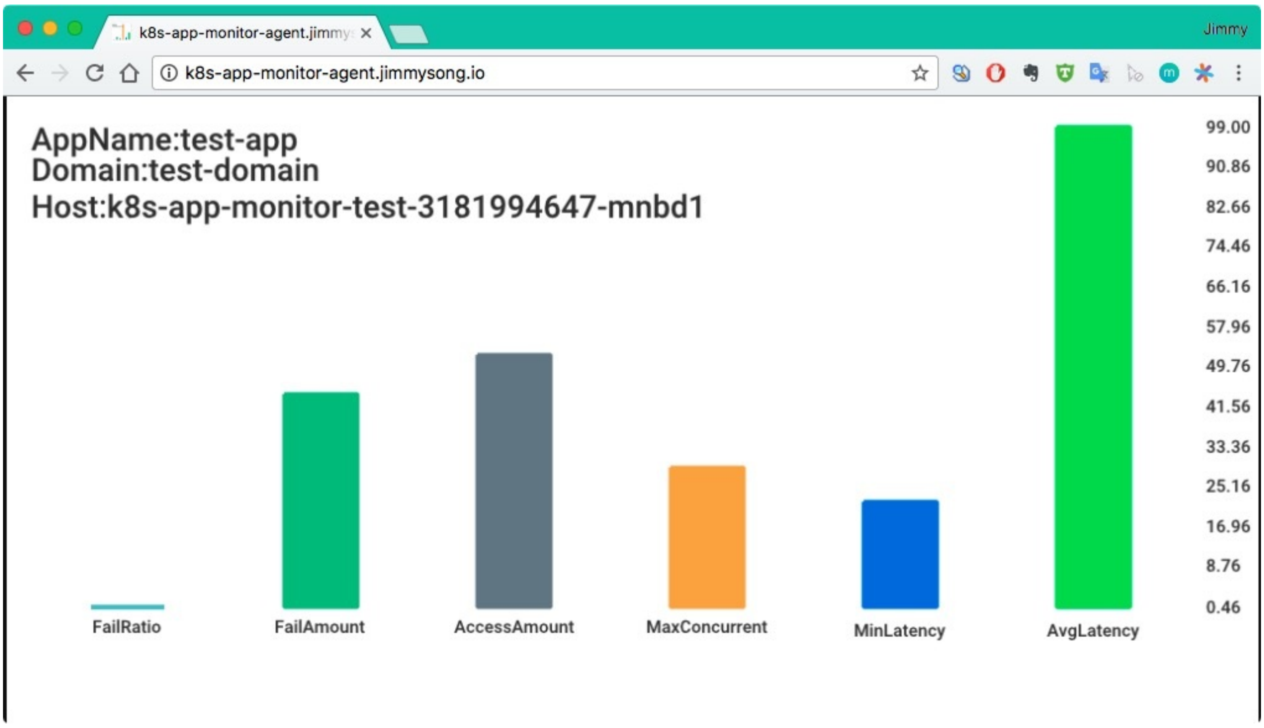
上一步中我们生成了kubernetes可读取的应用的YAML配置文件，我们可以将所有的YAML配置和并到同一个YAML文件中假如文件名为 `k8s-app-monitor-istio-all-in-one.yaml` ，如果要将其集成到Istio service mesh，只需要执行下面的命令。

```
kubectl apply -n default -f <(istioctl kube-inject -f k8s-app-mo
nitor-istio-all-in-one.yaml)
```

这样就会在每个Pod中注入一个sidecar容器。

验证

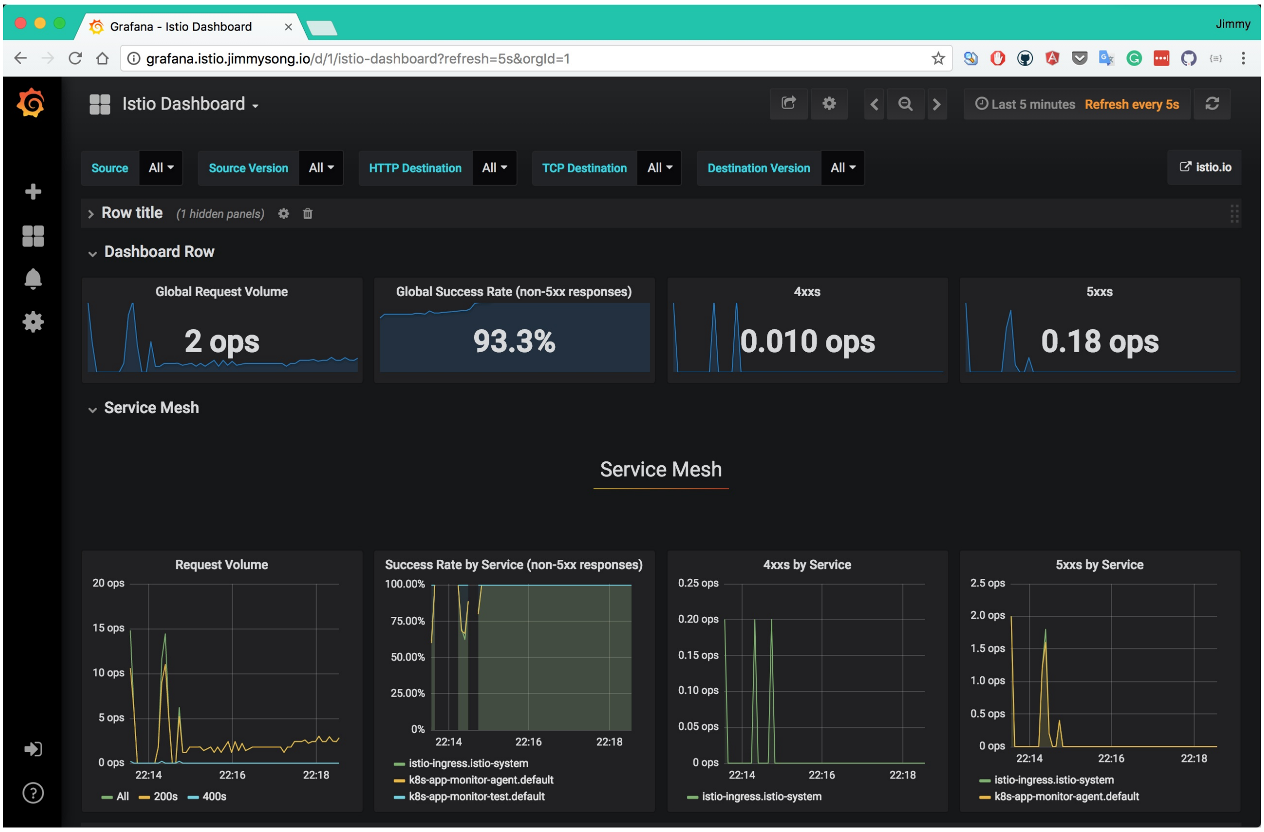
如果您使用的是Traefik ingress来暴露的服务，那么在浏览器中访问http://k8s-app-monitor-agent.jimmysong.io/k8s-app-monitor-agent，可以看到如下的画面，每次刷新页面将看到新的柱状图。



图片 - 图表

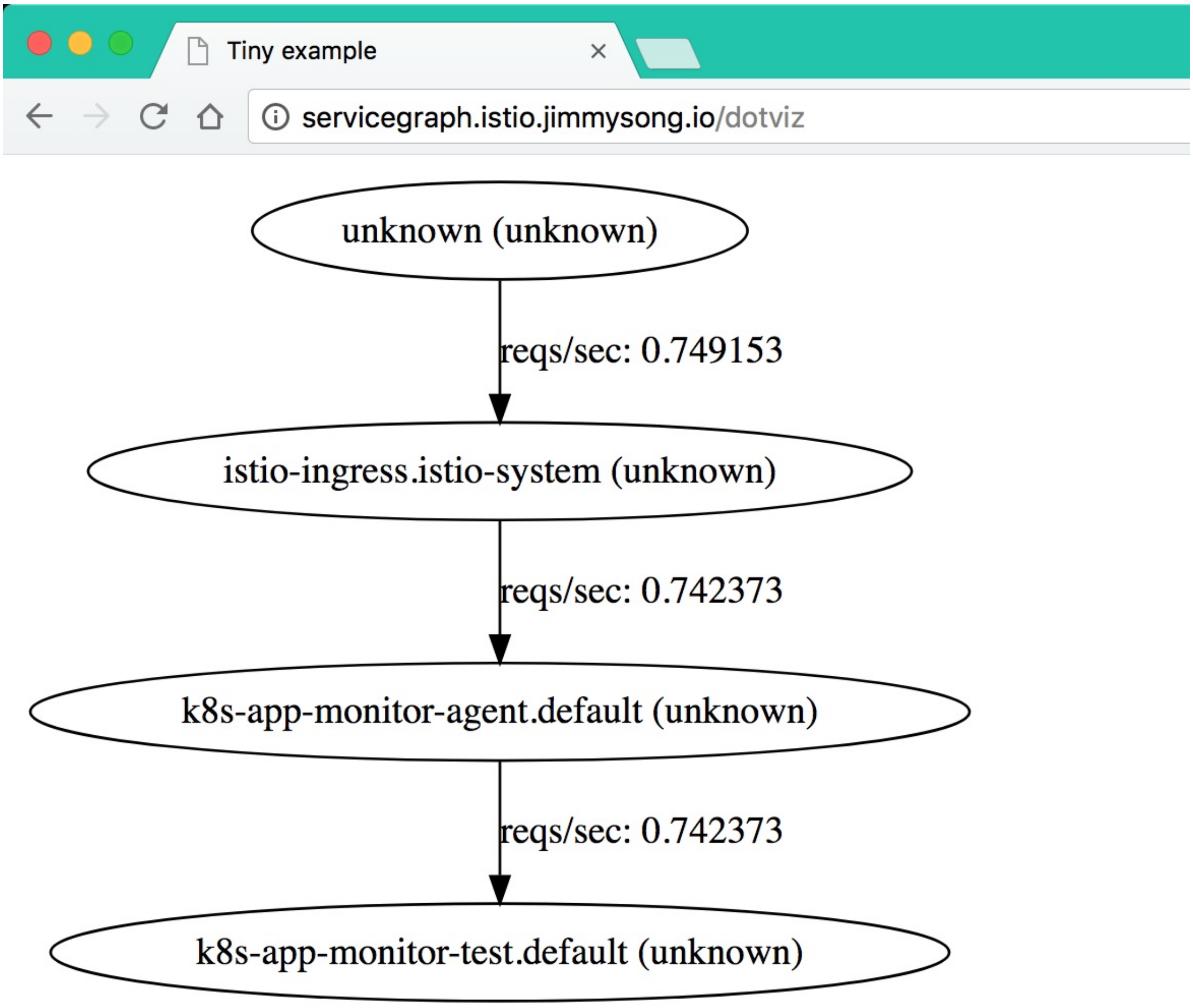
使用kubernetes-vagrant-centos-cluster来部署的kubernetes集群，该应用集成了Istio service mesh后可以通过http://172.17.8.101:32000/k8s-app-monitor-agent来访问。

在对*k8s-app-monitor-agent*服务进行了N此访问之后，再访问http://grafana.istio.jimmysong.io可以看到Service Mesh的监控信息。



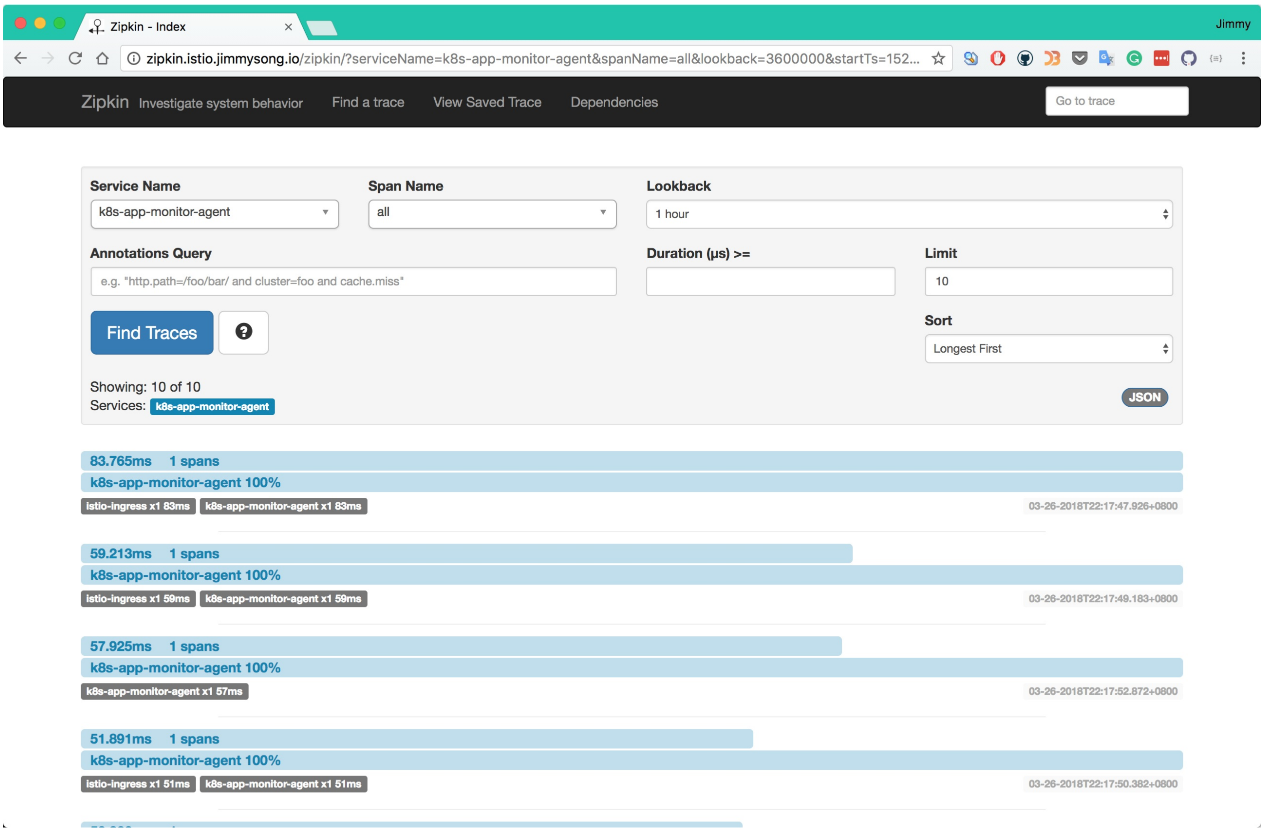
图片 - *Grafana*页面

访问http://servicegraph.istio.jimmysong.io/dotviz可以看到服务的依赖和QPS信息。



图片 - *servicegraph*页面

访问http://zipkin.istio.jimmysong.io可以选择查看 `k8s-app-monitor-agent` 应用的追踪信息。



图片 - *Zipkin*页面

至此从代码提交到上线到Kubernetes集群上并集成Istio service mesh的过程就全部完成了。

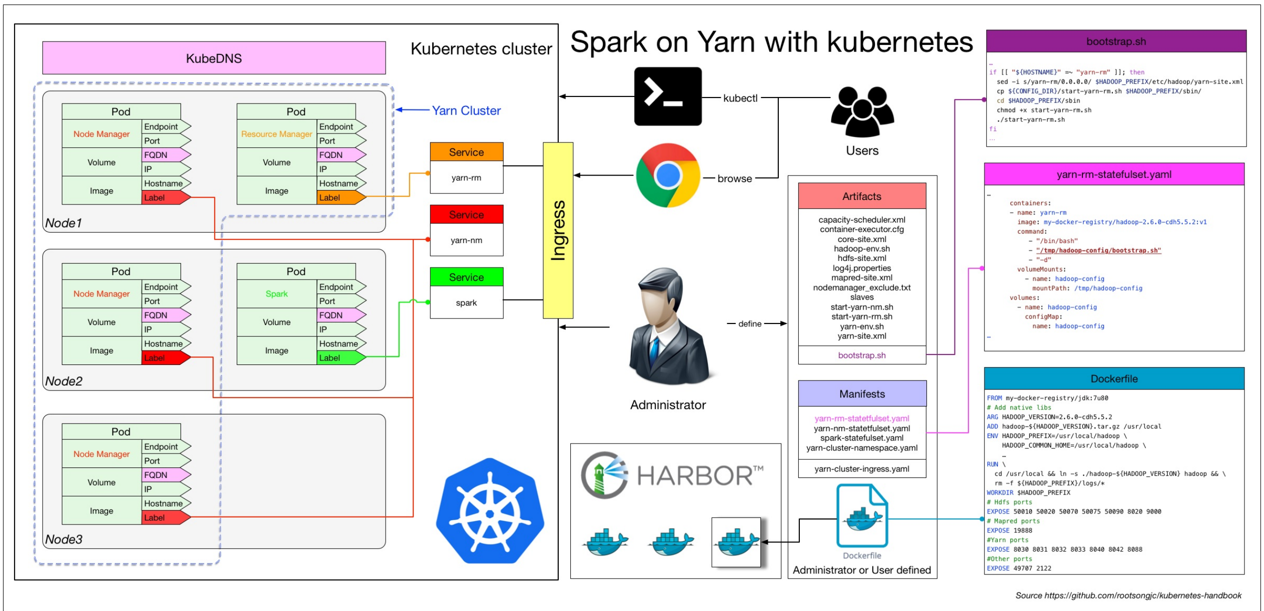
迁移传统应用到Kubernetes步骤详解——以Hadoop YARN为例

本文档不是说明如何在 kubernetes 中开发和部署应用程序，如果您想要直接开发应用程序在 kubernetes 中运行可以参考 适用于kubernetes的应用开发部署流程。

本文旨在说明如何将已有的应用程序尤其是传统的分布式应用程序迁移到 kubernetes 中。如果该类应用程序符合云原生应用规范（如12因素法则）的话，那么迁移会比较顺利，否则会遇到一些麻烦甚至是阻碍。具体请参考 迁移至云原生应用架构。

接下来我们将以 Spark on YARN with kubernetes 为例来说明，该例子足够复杂也很有典型性，了解了这个例子可以帮助大家将自己的应用迁移到 kubernetes 集群上去，代码和配置文件可以在 这里 找到（本文中加入 Spark 的配置，代码中并没有包含，读者可以自己配置）。

下图即整个架构的示意图，代码和详细配置文件请参考 kube-yarn（不包含 ingress、spark 配置），所有的进程管理和容器扩容直接使用 Makefile。



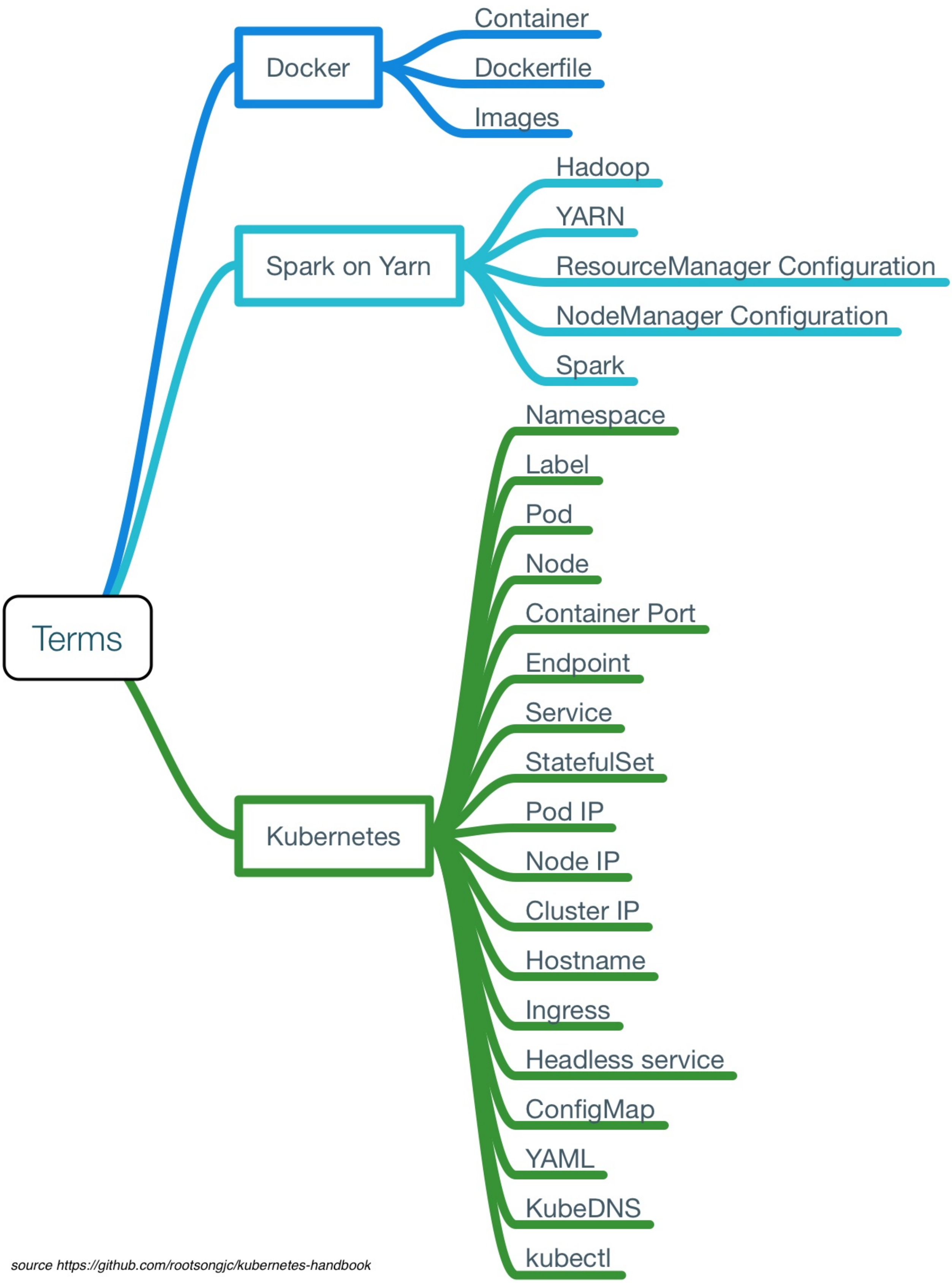
图片 - *spark on yarn with kubernetes*

注意：该例子仅用来说明具体的步骤划分和复杂性，在生产环境应用还有待验证，请谨慎使用。

术语

对于为曾接触过 kubernetes 或对云平台的技术细节不太了解的人来说，如何将应用迁移到 kubernetes 中可能是个头疼的问题，在行动之前有必要先了解整个过程中需要用到哪些概念和术语，有助于大家在行动中达成共识。

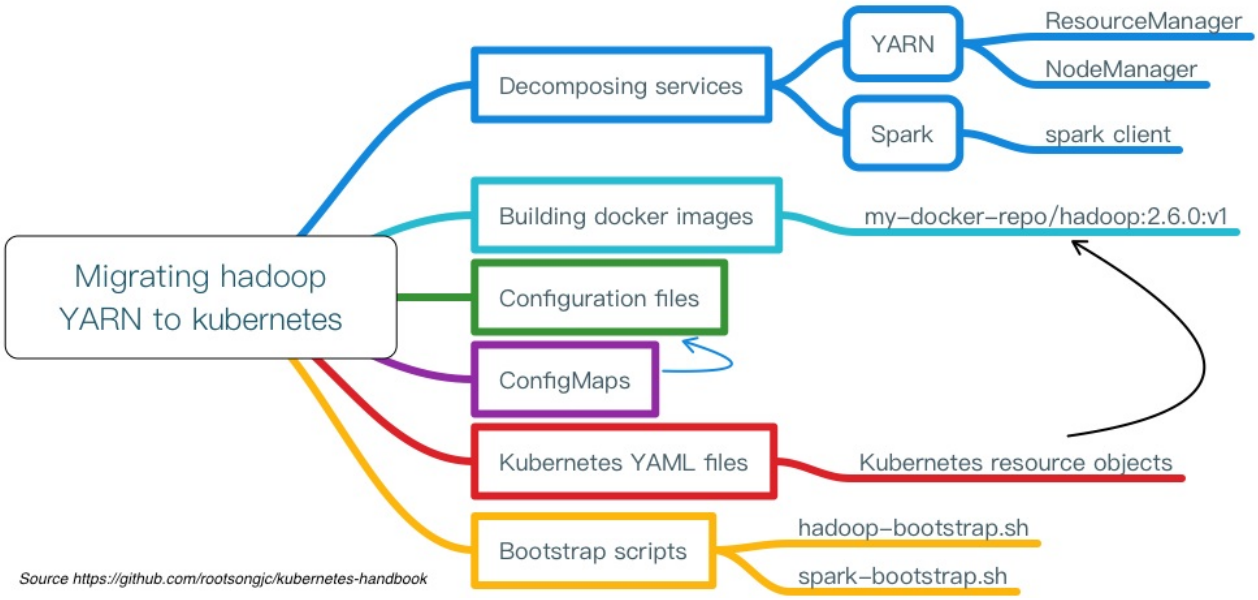
过程中可能用到的概念和术语初步整理如下：


Terms
Docker
Container
Dockerfile
Images
Spark on Yarn
Hadoop
YARN
ResourceManager Configuration
NodeManager Configuration
Spark
Kubernetes
Namespace
Label
Pod
Node
Container Port
Endpoint
Service
StatefulSet
Pod IP
Node IP
Cluster IP
Hostname
Ingress
Headless service
ConfigMap
YAML
KubeDNS
kubectl
source https://github.com/rootsongjc/kubernetes-handbook

图片 - *Terms*

为了讲解整改过程和具体细节，我们所有操作都是通过命令手动完成，不使用自动化工具。当您充分了解到其中的细节后可以通过自动化工具来优化该过程，以使其更加自动和高效，同时减少因为人为操作失误导致的迁移失败。

迁移应用



图片 - 分解步骤解析

整个迁移过程分为如下几个步骤：

1. 将原有应用拆解为服务

   我们不是一上来就开始做镜像，写配置，而是应该先梳理下要迁移的应用中有哪些可以作为服务运行，哪些是变的，哪些是不变的部分。

   服务划分的原则是最小可变原则，这个同样适用于镜像制作，将服务中不变的部分编译到同一个镜像中。

   对于像 Spark on YARN 这样复杂的应用，可以将其划分为三大类服务：

   - ResourceManager
   - NodeManager
   - Spark client
2. 制作镜像

根据拆解出来的服务，我们需要制作两个镜像：

- Hadoop
- Spark (From hadoop docker image)

因为我们运行的是 Spark on YARN，因此 Spark 依赖与 Hadoop 镜像，我们在 Spark 的基础上包装了一个 web service 作为服务启动。

镜像制作过程中不需要在 Dockerfile 中指定 Entrypoint 和 CMD，这些都是在 kubernetes 的 YAML 文件中指定的。

Hadoop YARN 的 Dockerfile 参考如下配置。

```
FROM my-docker-repo/jdk:7u80

# Add native libs
ARG HADOOP_VERSION=2.6.0-cdh5.5.2
## Prefer to download from server not use local storage
ADD hadoop-${HADOOP_VERSION}.tar.gz /usr/local
ADD ./lib/* /usr/local/hadoop-${HADOOP_VERSION}/lib/native/
ADD ./jars/* /usr/local/hadoop-${HADOOP_VERSION}/share/hadoo
p/yarn/
ENV HADOOP_PREFIX=/usr/local/hadoop \
    HADOOP_COMMON_HOME=/usr/local/hadoop \
    HADOOP_HDFS_HOME=/usr/local/hadoop \
    HADOOP_MAPRED_HOME=/usr/local/hadoop \
    HADOOP_YARN_HOME=/usr/local/hadoop \
    HADOOP_CONF_DIR=/usr/local/hadoop/etc/hadoop \
    YARN_CONF_DIR=/usr/local/hadoop/etc/hadoop \
    PATH=${PATH}:/usr/local/hadoop/bin

RUN \
  cd /usr/local && ln -s ./hadoop-${HADOOP_VERSION} hadoop &
& \
  rm -f ${HADOOP_PREFIX}/logs/*

WORKDIR $HADOOP_PREFIX

# Hdfs ports
EXPOSE 50010 50020 50070 50075 50090 8020 9000
# Mapred ports
EXPOSE 19888
#Yarn ports
EXPOSE 8030 8031 8032 8033 8040 8042 8088
#Other ports
EXPOSE 49707 2122
```

3. 准备应用的配置文件

因为我们只制作了一个 Hadoop 的镜像，而需要启动两个服务，这就要求在服务启动的时候必须加载不同的配置文件，现在我们只需要准备两个服务中需要同时用的的配置的部分。

YARN 依赖的配置在 `artifacts` 目录下，包含以下文件：

```
bootstrap.sh
capacity-scheduler.xml
container-executor.cfg
core-site.xml
hadoop-env.sh
hdfs-site.xml
log4j.properties
mapred-site.xml
nodemanager_exclude.txt
slaves
start-yarn-nm.sh
start-yarn-rm.sh
yarn-env.sh
yarn-site.xml
```

其中作为 bootstrap 启动脚本的 `bootstrap.sh` 也包含在该目录下，该脚本的如何编写请见下文。

4. **Kubernetes YAML** 文件

   根据业务的特性选择最适合的 kubernetes 的资源对象来运行，因为在 YARN 中 NodeManager 需要使用主机名向 ResourceManger 注册，因此需要沿用 YARN 原有的服务发现方式，使用 headless service 和 StatefulSet 资源。更多资料请参考 StatefulSet。

   所有的 Kubernetes YAML 配置文件存储在 `manifest` 目录下，包括如下配置：

   - yarn-cluster 的 namespace 配置
   - Spark、ResourceManager、NodeManager 的 headless service 和 StatefulSet 配置
   - 需要暴露到 kubernetes 集群外部的 ingress 配置（ResourceManager 的 Web）

```
kube-yarn-ingress.yaml
spark-statefulset.yaml
yarn-cluster-namespace.yaml
yarn-nm-statefulset.yaml
yarn-rm-statefulset.yaml
```

5. **Bootstrap** 脚本

   Bootstrap 脚本的作用是在启动时根据 Pod 的环境变量、主机名或其他可以区分不同 Pod 和将启动角色的变量来修改配置文件和启动服务应用。

   该脚本同时将原来 YARN 的日志使用 stdout 输出，便于使用 `kubectl logs` 查看日志或其他日志收集工具进行日志收集。

   启动脚本 `bootstrap.sh` 跟 Hadoop 的配置文件同时保存在 `artifacts` 目录下。

   该脚本根据 Pod 的主机名，决定如何修改 Hadoop 的配置文件和启动何种服务。`bootstrap.sh` 文件的部分代码如下：

```
if [[ "${HOSTNAME}" =~ "yarn-nm" ]]; then
  sed -i '/<\/configuration>/d' $HADOOP_PREFIX/etc/hadoop/ya
rn-site.xml
  cat >> $HADOOP_PREFIX/etc/hadoop/yarn-site.xml <<- EOM
  <property>
    <name>yarn.nodemanager.resource.memory-mb</name>
    <value>${MY_MEM_LIMIT:-2048}</value>
  </property>

  <property>
    <name>yarn.nodemanager.resource.cpu-vcores</name>
    <value>${MY_CPU_LIMIT:-2}</value>
  </property>
EOM
  echo '</configuration>' >> $HADOOP_PREFIX/etc/hadoop/yarn-
site.xml
  cp ${CONFIG_DIR}/start-yarn-nm.sh $HADOOP_PREFIX/sbin/
  cd $HADOOP_PREFIX/sbin
  chmod +x start-yarn-nm.sh
  ./start-yarn-nm.sh
fi

if [[ $1 == "-d" ]]; then
  until find ${HADOOP_PREFIX}/logs -mmin -1 | egrep -q '.*';
echo "`date`: Waiting for logs..." ; do sleep 2 ; done
  tail -F ${HADOOP_PREFIX}/logs/* &
  while true; do sleep 1000; done
fi
```

从这部分中代码中可以看到，如果 Pod 的主机名中包含 `yarn-nm` 字段则向 `yarn-site.xml` 配置文件中增加如下内容：

```
  <property>
    <name>yarn.nodemanager.resource.memory-mb</name>
    <value>${MY_MEM_LIMIT:-2048}</value>
  </property>

  <property>
    <name>yarn.nodemanager.resource.cpu-vcores</name>
    <value>${MY_CPU_LIMIT:-2}</value>
  </property>
```

其中 `MY_MEM_LIMIT` 和 `MY_CPU_LIMIT` 是 kubernetes YAML 中定义的环境变量，该环境变量又是引用的 Resource limit。

所有的配置准备完成后，执行 `start-yarn-nm.sh` 脚本启动 NodeManager。

如果 kubernetes YAML 中的 container CMD args 中包含 `-d` 则在后台运行 NodeManger 并 tail 输出 NodeManager 的日志到标准输出。

6. **ConfigMaps**

   将 Hadoop 的配置文件和 bootstrap 脚本作为 ConfigMap 资源保存，用作 Pod 启动时挂载的 volume。

```
kubectl create configmap hadoop-config \
      --from-file=artifacts/hadoop/bootstrap.sh \
      --from-file=artifacts/hadoop/start-yarn-rm.sh \
      --from-file=artifacts/hadoop/start-yarn-nm.sh \
      --from-file=artifacts/hadoop/slaves \
      --from-file=artifacts/hadoop/core-site.xml \
      --from-file=artifacts/hadoop/hdfs-site.xml \
      --from-file=artifacts/hadoop/mapred-site.xml \
      --from-file=artifacts/hadoop/yarn-site.xml \
      --from-file=artifacts/hadoop/capacity-scheduler.xml \
      --from-file=artifacts/hadoop/container-executor.cfg \
      --from-file=artifacts/hadoop/hadoop-env.sh \
      --from-file=artifacts/hadoop/log4j.properties \
      --from-file=artifacts/hadoop/nodemanager_exclude.txt \
      --from-file=artifacts/hadoop/yarn-env.sh
kubectl  create configmap spark-config \
      --from-file=artifacts/spark/spark-bootstrap.sh \
      --from-file=artifacts/spark/spark-env.sh \
      --from-file=artifacts/spark/spark-defaults.conf
```

所有的配置完成后，可以可以使用 kubectl 命令来启动和管理集群了，我们编写了 Makefile，您可以直接使用该 Makefile 封装的命令实现部分的自动化。

使用StatefulSet部署有状态应用

StatefulSet 这个对象是专门用来部署用状态应用的，可以为Pod提供稳定的身份标识，包括hostname、启动顺序、DNS名称等。

下面以在kubernetes1.6版本中部署zookeeper和kafka为例讲解StatefulSet的使用，其中kafka依赖于zookeeper。

Dockerfile和配置文件见 zookeeper 和 kafka。

注：所有的镜像基于CentOS系统的JDK制作，为我的私人镜像，外部无法访问，yaml中没有配置持久化存储。

部署Zookeeper

Dockerfile中从远程获取zookeeper的安装文件，然后在定义了三个脚本：

- zkGenConfig.sh：生成zookeeper配置文件
- zkMetrics.sh：获取zookeeper的metrics
- zkOk.sh：用来做ReadinessProb

我们在来看下这三个脚本的执行结果：

zkGenConfig.sh

zkMetrics.sh脚本实际上执行的是下面的命令：

```
$ echo mntr | nc localhost $ZK_CLIENT_PORT >& 1
zk_version    3.4.6-1569965, built on 02/20/2014 09:09 GMT
zk_avg_latency    0
zk_max_latency    5
zk_min_latency    0
zk_packets_received    427879
zk_packets_sent    427890
zk_num_alive_connections    3
zk_outstanding_requests    0
zk_server_state    leader
zk_znode_count    18
zk_watch_count    3
zk_ephemerals_count    4
zk_approximate_data_size    613
zk_open_file_descriptor_count    29
zk_max_file_descriptor_count    1048576
zk_followers    1
zk_synced_followers    1
zk_pending_syncs    0
```

zkOk.sh脚本实际上执行的是下面的命令：

```
$ echo ruok | nc 127.0.0.1 $ZK_CLIENT_PORT
imok
```

zookeeper.yaml

下面是启动三个zookeeper实例的yaml配置文件：

```
---
apiVersion: v1
kind: Service
metadata:
  name: zk-svc
  labels:
    app: zk-svc
spec:
  ports:
  - port: 2888
    name: server
  - port: 3888
    name: leader-election
  clusterIP: None
  selector:
    app: zk
---
apiVersion: v1
kind: ConfigMap
```

```yaml
metadata:
  name: zk-cm
data:
  jvm.heap: "1G"
  tick: "2000"
  init: "10"
  sync: "5"
  client.cnxns: "60"
  snap.retain: "3"
  purge.interval: "0"
---
apiVersion: policy/v1beta1
kind: PodDisruptionBudget
metadata:
  name: zk-pdb
spec:
  selector:
    matchLabels:
      app: zk
  minAvailable: 2
---
apiVersion: apps/v1beta1
kind: StatefulSet
metadata:
  name: zk
spec:
  serviceName: zk-svc
  replicas: 3
  template:
    metadata:
      labels:
        app: zk
    spec:
      affinity:
        podAntiAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            - labelSelector:
                matchExpressions:
                  - key: "app"
                    operator: In
                    values:
                    - zk
              topologyKey: "kubernetes.io/hostname"
      containers:
      - name: k8szk
        imagePullPolicy: Always
        image: harbor-001.jimmysong.io/library/zookeeper:3.4.6
        resources:
          requests:
            memory: "2Gi"
            cpu: "500m"
        ports:
        - containerPort: 2181
```

```
        name: client
      - containerPort: 2888
        name: server
      - containerPort: 3888
        name: leader-election
      env:
      - name : ZK_REPLICAS
        value: "3"
      - name : ZK_HEAP_SIZE
        valueFrom:
          configMapKeyRef:
              name: zk-cm
              key: jvm.heap
      - name : ZK_TICK_TIME
        valueFrom:
          configMapKeyRef:
              name: zk-cm
              key: tick
      - name : ZK_INIT_LIMIT
        valueFrom:
          configMapKeyRef:
              name: zk-cm
              key: init
      - name : ZK_SYNC_LIMIT
        valueFrom:
          configMapKeyRef:
              name: zk-cm
              key: tick
      - name : ZK_MAX_CLIENT_CNXNS
        valueFrom:
          configMapKeyRef:
              name: zk-cm
              key: client.cnxns
      - name: ZK_SNAP_RETAIN_COUNT
        valueFrom:
          configMapKeyRef:
              name: zk-cm
              key: snap.retain
      - name: ZK_PURGE_INTERVAL
        valueFrom:
          configMapKeyRef:
              name: zk-cm
              key: purge.interval
      - name: ZK_CLIENT_PORT
        value: "2181"
      - name: ZK_SERVER_PORT
        value: "2888"
      - name: ZK_ELECTION_PORT
        value: "3888"
      command:
      - sh
      - -c
      - zkGenConfig.sh && zkServer.sh start-foreground
```

```
        readinessProbe:
          exec:
            command:
            - "zkOk.sh"
          initialDelaySeconds: 10
          timeoutSeconds: 5
        livenessProbe:
          exec:
            command:
            - "zkOk.sh"
          initialDelaySeconds: 10
          timeoutSeconds: 5
      securityContext:
        runAsUser: 1000
        fsGroup: 1000
```

我们再主要下上面那三个脚本的用途。

部署kafka

Kafka的docker镜像制作跟zookeeper类似，都是从远程下载安装包后，解压安装。

与zookeeper不同的是，只要一个脚本，但是又依赖于我们上一步安装的zookeeper，kafkaGenConfig.sh用来生成kafka的配置文件。

我们来看下这个脚本。

```
#!/bin/bash
HOST=`hostname -s`
if [[ $HOST =~ (.*)-([0-9]+)$ ]]; then
        NAME=${BASH_REMATCH[1]}
        ORD=${BASH_REMATCH[2]}
else
        echo "Failed to extract ordinal from hostname $HOST"
        exit 1
fi

MY_ID=$((ORD+1))
sed -i s"/broker.id=0/broker.id=$MY_ID/g" /opt/kafka/config/serv
er.properties
sed -i s'/zookeeper.connect=localhost:2181/zookeeper.connect=zk-
0.zk-svc.brand.svc:2181,zk-1.zk-svc.brand.svc:2181,zk-2.zk-svc.b
rand.svc:2181/g' /opt/kafka/config/server.properties
```

该脚本根据statefulset生成的pod的hostname的后半截数字部分作为broker ID，同时再替换zookeeper的地址。

Kafka.yaml

下面是创建3个kafka实例的yaml配置。

```yaml
---
apiVersion: v1
kind: Service
metadata:
  name: kafka-svc
  labels:
    app: kafka
spec:
  ports:
  - port: 9093
    name: server
  clusterIP: None
  selector:
    app: kafka
---
apiVersion: policy/v1beta1
kind: PodDisruptionBudget
metadata:
  name: kafka-pdb
spec:
  selector:
    matchLabels:
      app: kafka
  minAvailable: 2
---
apiVersion: apps/v1beta1
kind: StatefulSet
metadata:
  name: kafka
spec:
  serviceName: kafka-svc
  replicas: 3
  template:
    metadata:
      labels:
        app: kafka
    spec:
      affinity:
        podAntiAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            - labelSelector:
                matchExpressions:
                  - key: "app"
                    operator: In
                    values:
                    - kafka
              topologyKey: "kubernetes.io/hostname"
        podAffinity:
```

```
          preferredDuringSchedulingIgnoredDuringExecution:
            - weight: 1
              podAffinityTerm:
                labelSelector:
                   matchExpressions:
                     - key: "app"
                       operator: In
                       values:
                       - zk
                topologyKey: "kubernetes.io/hostname"
      terminationGracePeriodSeconds: 300
      containers:
      - name: k8skafka
        imagePullPolicy: Always
        image: harbor-001.jimmysong.io/library/kafka:2.10-0.8.2.
1
        resources:
          requests:
            memory: "1Gi"
            cpu: 500m
        env:
        - name: KF_REPLICAS
          value: "3"
        ports:
        - containerPort: 9093
          name: server
        command:
        - /bin/bash
        - -c
        - "/opt/kafka/bin/kafkaGenConfig.sh && /opt/kafka/bin/ka
fka-server-start.sh /opt/kafka/config/server.properties"
        env:
        - name: KAFKA_HEAP_OPTS
          value : "-Xmx512M -Xms512M"
        - name: KAFKA_OPTS
          value: "-Dlogging.level=DEBUG"
        readinessProbe:
           tcpSocket:
             port: 9092
           initialDelaySeconds: 15
           timeoutSeconds: 1
```

参考

- https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/
- kubernetes contrib - statefulsets
- http://blog.kubernetes.io/2017/01/running-mongodb-on-kubernetes-with-statefulsets.html

最佳实践概览

本章节从零开始创建我们自己的kubernetes集群，并在该集群的基础上，配置服务发现、负载均衡和日志收集等功能，使我们的集群能够成为一个真正线上可用、功能完整的集群。

- 第一部分 在CentOS上部署kubernetes集群中介绍了如何通过二进制文件在CentOS物理机（也可以是公有云主机）上快速部署一个kubernetes集群。
- 第二部分介绍如何在kubernetes中的服务发现与负载均衡。
- 第三部分介绍如何运维kubernetes集群。
- 第四部分介绍kubernetes中的存储管理。
- 第五部分关于kubernetes集群和应用的监控。
- 第六部分介绍kuberentes中的服务编排与管理。
- 第七部分介绍如何基于kubernetes做持续集成与发布。
- 第八部分是kubernetes集群与插件的更新升级。

在CentOS上部署kubernetes集群

> 本文档最初是基于kubenetes1.6版本编写的，对于kuberentes1.8及以上版本同样适用，只是个别位置有稍许变动，变动的地方我将特别注明版本要求。

本系列文档介绍使用二进制部署 `kubernetes` 集群的所有步骤，而不是使用 `kubeadm` 等自动化方式来部署集群，同时开启了集群的TLS安全认证，该安装步骤适用于所有bare metal环境、on-premise环境和公有云环境。

> 如果您想快速的在自己电脑的本地环境下使用虚拟机来搭建kubernetes集群，可以参考本地分布式开发环境搭建（使用Vagrant和Virtualbox）。

在部署的过程中，将详细列出各组件的启动参数，给出配置文件，详解它们的含义和可能遇到的问题。

部署完成后，你将理解系统各组件的交互原理，进而能快速解决实际问题。

所以本文档主要适合于那些有一定 kubernetes 基础，想通过一步步部署的方式来学习和了解系统配置、运行原理的人。

注：本文档中不包括**docker**和私有镜像仓库的安装，安装说明中使用的镜像来自 **Google Cloud Platform**，为了方便国内用户下载，我将其克隆并上传到了 时速云镜像市场，供大家免费下载。

欲下载最新版本的官方镜像请访问 **Google** 云平台容器注册表。

提供所有的配置文件

集群安装时所有组件用到的配置文件，包含在以下目录中：

- **etc**：service的环境变量配置文件
- **manifest**：kubernetes应用的yaml文件
- **systemd** ：systemd serivce配置文件

集群详情

- OS：CentOS Linux release 7.3.1611 (Core) 3.10.0-514.16.1.el7.x86_64

- Kubernetes 1.6.0+（最低的版本要求是1.6）
- Docker 1.12.5（使用yum安装）
- Etcd 3.1.5
- Flannel 0.7.1 vxlan或者host-gw 网络
- TLS 认证通信 (所有组件，如 etcd、kubernetes master 和 node)
- RBAC 授权
- kubelet TLS BootStrapping
- kubedns、dashboard、heapster(influxdb、grafana)、EFK(elasticsearch、fluentd、kibana) 集群插件
- 私有docker镜像仓库harbor（请自行部署，harbor提供离线安装包，直接使用docker-compose启动即可）

环境说明

在下面的步骤中，我们将在三台CentOS系统的物理机上部署具有三个节点的kubernetes1.6.0集群。

角色分配如下：

镜像仓库：172.20.0.112，域名为 `harbor.jimmysong.io` ，为私有镜像仓库，请替换为公共仓库或你自己的镜像仓库地址。

Master：172.20.0.113

Node：172.20.0.113、172.20.0.114、172.20.0.115

注意：172.20.0.113这台主机master和node复用。所有生成证书、执行kubectl命令的操作都在这台节点上执行。一旦node加入到kubernetes集群之后就不需要再登陆node节点了。

安装前的准备

1. 在node节点上安装docker1.12.5

   直接使用 `yum install docker`

2. 关闭所有节点的SELinux

   永久方法 – 需要重启服务器

修改 `/etc/selinux/config` 文件中设置SELINUX=disabled ，然后重启服务器。

临时方法 – 设置系统参数

使用命令 `setenforce 0`

附： setenforce 1 设置SELinux 成为enforcing模式 setenforce 0 设置SELinux 成为permissive模式

3. 准备harbor私有镜像仓库

   参考：https://github.com/vmware/harbor

步骤介绍

1. 创建 TLS 证书和秘钥
2. 创建kubeconfig 文件
3. 创建高可用etcd集群
4. 安装kubectl命令行工具
5. 部署master节点
6. 安装flannel网络插件
7. 部署node节点
8. 安装kubedns插件
9. 安装dashboard插件
10. 安装heapster插件
11. 安装EFK插件

提醒

1. 由于启用了 TLS 双向认证、RBAC 授权等严格的安全机制，建议从头开始部署，而不要从中间开始，否则可能会认证、授权等失败！
2. 部署过程中需要有很多证书的操作，请大家耐心操作，不明白的操作可以参考本书中的其他章节的解释。
3. 该部署操作仅是搭建成了一个可用 kubernetes 集群，而很多地方还需要进行优化，heapster 插件、EFK 插件不一定会用于真实的生产环境中，但是通过部署这些插件，可以让大家了解到如何部署应用到集群上。

注：本安装文档参考了 opsnull 跟我一步步部署 kubernetes 集群

创建TLS证书和秘钥

前言

执行下列步骤前建议你先阅读以下内容：

- 管理集群中的TLS：教您如何创建TLS证书
- kubelet的认证授权：向您描述如何通过认证授权来访问 kubelet 的 HTTPS 端点。
- TLS bootstrap：介绍如何为 kubelet 设置 TLS 客户端证书引导（bootstrap）。

注意：这一步是在安装配置kubernetes的所有步骤中最容易出错也最难于排查问题的一步，而这却刚好是第一步，万事开头难，不要因为这点困难就望而却步。

如果您足够有信心在完全不了解自己在做什么的情况下能够成功地完成了这一步的配置，那么您可以尽管跳过上面的几篇文章直接进行下面的操作。

`kubernetes` 系统的各组件需要使用 `TLS` 证书对通信进行加密，本文档使用 `CloudFlare` 的 PKI 工具集 cfssl 来生成 Certificate Authority (CA) 和其它证书；

生成的 **CA** 证书和秘钥文件如下：

- ca-key.pem
- ca.pem
- kubernetes-key.pem
- kubernetes.pem
- kube-proxy.pem
- kube-proxy-key.pem
- admin.pem
- admin-key.pem

使用证书的组件如下：

- etcd：使用 ca.pem、kubernetes-key.pem、kubernetes.pem；
- kube-apiserver：使用 ca.pem、kubernetes-key.pem、kubernetes.pem；
- kubelet：使用 ca.pem；
- kube-proxy：使用 ca.pem、kube-proxy-key.pem、kube-proxy.pem；

- kubectl：使用 ca.pem、admin-key.pem、admin.pem；
- kube-controller-manager：使用 ca-key.pem、ca.pem

注意：以下操作都在 **master** 节点即 **172.20.0.113** 这台主机上执行，证书只需要创建一次即可，以后在向集群中添加新节点时只要将 **/etc/kubernetes/** 目录下的证书拷贝到新节点上即可。

安装 `CFSSL`

方式一：直接使用二进制源码包安装

```
wget https://pkg.cfssl.org/R1.2/cfssl_linux-amd64
chmod +x cfssl_linux-amd64
mv cfssl_linux-amd64 /usr/local/bin/cfssl

wget https://pkg.cfssl.org/R1.2/cfssljson_linux-amd64
chmod +x cfssljson_linux-amd64
mv cfssljson_linux-amd64 /usr/local/bin/cfssljson

wget https://pkg.cfssl.org/R1.2/cfssl-certinfo_linux-amd64
chmod +x cfssl-certinfo_linux-amd64
mv cfssl-certinfo_linux-amd64 /usr/local/bin/cfssl-certinfo

export PATH=/usr/local/bin:$PATH
```

方式二：使用go命令安装

我们的系统中安装了Go1.7.5，使用以下命令安装更快捷：

```
$ go get -u github.com/cloudflare/cfssl/cmd/...
$ echo $GOPATH
/usr/local
$ls /usr/local/bin/cfssl*
cfssl cfssl-bundle cfssl-certinfo cfssljson cfssl-newkey cfssl-s
can
```

在 `$GOPATH/bin` 目录下得到以cfssl开头的几个命令。

注意：以下文章中出现的cat的文件名如果不存在需要手工创建。

创建 CA (Certificate Authority)

创建 **CA** 配置文件

```
mkdir /root/ssl
cd /root/ssl
cfssl print-defaults config > config.json
cfssl print-defaults csr > csr.json
# 根据config.json文件的格式创建如下的ca-config.json文件
# 过期时间设置成了 87600h
cat > ca-config.json <<EOF
{
  "signing": {
    "default": {
      "expiry": "87600h"
    },
    "profiles": {
      "kubernetes": {
        "usages": [
            "signing",
            "key encipherment",
            "server auth",
            "client auth"
        ],
        "expiry": "87600h"
      }
    }
  }
}
EOF
```

字段说明

- `ca-config.json` ：可以定义多个 profiles，分别指定不同的过期时间、使用场景等参数；后续在签名证书时使用某个 profile；
- `signing` ：表示该证书可用于签名其它证书；生成的 ca.pem 证书中 `CA=TRUE` ；
- `server auth` ：表示client可以用该 CA 对server提供的证书进行验证；
- `client auth` ：表示server可以用该CA对client提供的证书进行验证；

创建 **CA** 证书签名请求

创建 `ca-csr.json` 文件，内容如下：

```
{
  "CN": "kubernetes",
  "key": {
    "algo": "rsa",
    "size": 2048
  },
  "names": [
    {
      "C": "CN",
      "ST": "BeiJing",
      "L": "BeiJing",
      "O": "k8s",
      "OU": "System"
    }
  ],
    "ca": {
       "expiry": "87600h"
    }
}
```

- "CN"： `Common Name` ，kube-apiserver 从证书中提取该字段作为请求的用户名 (User Name)；浏览器使用该字段验证网站是否合法；
- "O"： `Organization` ，kube-apiserver 从证书中提取该字段作为请求用户所属的组 (Group)；

生成 **CA** 证书和私钥

```
$ cfssl gencert -initca ca-csr.json | cfssljson -bare ca
$ ls ca*
ca-config.json  ca.csr  ca-csr.json  ca-key.pem  ca.pem
```

创建 kubernetes 证书

创建 kubernetes 证书签名请求文件 `kubernetes-csr.json` ：

```
{
    "CN": "kubernetes",
    "hosts": [
      "127.0.0.1",
      "172.20.0.112",
      "172.20.0.113",
      "172.20.0.114",
      "172.20.0.115",
      "10.254.0.1",
      "kubernetes",
      "kubernetes.default",
      "kubernetes.default.svc",
      "kubernetes.default.svc.cluster",
      "kubernetes.default.svc.cluster.local"
    ],
    "key": {
        "algo": "rsa",
        "size": 2048
    },
    "names": [
        {
            "C": "CN",
            "ST": "BeiJing",
            "L": "BeiJing",
            "O": "k8s",
            "OU": "System"
        }
    ]
}
```

- 如果 hosts 字段不为空则需要指定授权使用该证书的 **IP** 或域名列表，由于该证书后续被 `etcd` 集群和 `kubernetes master` 集群使用，所以上面分别指定了 `etcd` 集群、 `kubernetes master` 集群的主机 IP 和 **`kubernetes`** 服务的服务 **IP**（一般是 `kube-apiserver` 指定的 `service-cluster-ip-range` 网段的第一个IP，如 10.254.0.1）。
- 这是最小化安装的kubernetes集群，包括一个私有镜像仓库，三个节点的kubernetes集群，以上物理节点的IP也可以更换为主机名。

生成 **kubernetes** 证书和私钥

```
$ cfssl gencert -ca=ca.pem -ca-key=ca-key.pem -config=ca-config.
json -profile=kubernetes kubernetes-csr.json | cfssljson -bare k
ubernetes
$ ls kubernetes*
kubernetes.csr  kubernetes-csr.json  kubernetes-key.pem  kuberne
tes.pem
```

或者直接在命令行上指定相关参数：

```
echo '{"CN":"kubernetes","hosts":[""],"key":{"algo":"rsa","size"
:2048}}' | cfssl gencert -ca=ca.pem -ca-key=ca-key.pem -config=c
a-config.json -profile=kubernetes -hostname="127.0.0.1,172.20.0.
112,172.20.0.113,172.20.0.114,172.20.0.115,kubernetes,kubernetes
.default" - | cfssljson -bare kubernetes
```

创建 admin 证书

创建 admin 证书签名请求文件 `admin-csr.json` ：

```
{
  "CN": "admin",
  "hosts": [],
  "key": {
    "algo": "rsa",
    "size": 2048
  },
  "names": [
    {
      "C": "CN",
      "ST": "BeiJing",
      "L": "BeiJing",
      "O": "system:masters",
      "OU": "System"
    }
  ]
}
```

- 后续 `kube-apiserver` 使用 `RBAC` 对客户端(如 `kubelet` 、 `kube-proxy` 、 `Pod`)请求进行授权；
- `kube-apiserver` 预定义了一些 `RBAC` 使用的 `RoleBindings` ，如 `cluster-admin` 将 Group `system:masters` 与 Role `cluster-admin` 绑定，该 Role 授予了调用 `kube-apiserver` 的所有 **API**的权限；
- O 指定该证书的 Group 为 `system:masters` ， `kubelet` 使用该证书访问 `kube-apiserver` 时 ，由于证书被 CA 签名，所以认证通过，同时由于证书用户组为经过预授权的 `system:masters` ，所以被授予访问所有 API 的权限；

注意：这个admin 证书，是将来生成管理员用的kube config 配置文件用的，现在我们一般建议使用RBAC 来对kubernetes 进行角色权限控制， kubernetes 将证书中的CN 字段 作为User， O 字段作为 Group（具体参考 Kubernetes中的用户与身份认证授权中 X509 Client Certs 一段）。

在搭建完 kubernetes 集群后，我们可以通过命令: `kubectl get clusterrolebinding cluster-admin -o yaml` ,查看到 `clusterrolebinding cluster-admin` 的 subjects 的 kind 是 Group，name 是 `system:masters` 。 `roleRef` 对象是 `ClusterRole cluster-admin` 。 意思是凡是 `system:masters Group` 的 user 或者 `serviceAccount` 都拥有 `cluster-admin` 的角色。 因此我们在使用 kubectl 命令时候，才拥有整个集群的管理权限。可以使用 `kubectl get clusterrolebinding cluster-admin -o yaml` 来查看。

```
$ kubectl get clusterrolebinding cluster-admin -o yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  annotations:
    rbac.authorization.kubernetes.io/autoupdate: "true"
  creationTimestamp: 2017-04-11T11:20:42Z
  labels:
    kubernetes.io/bootstrapping: rbac-defaults
  name: cluster-admin
  resourceVersion: "52"
  selfLink: /apis/rbac.authorization.k8s.io/v1/clusterrolebindin
gs/cluster-admin
  uid: e61b97b2-1ea8-11e7-8cd7-f4e9d49f8ed0
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: cluster-admin
subjects:
- apiGroup: rbac.authorization.k8s.io
  kind: Group
  name: system:masters
```

生成 admin 证书和私钥：

```
$ cfssl gencert -ca=ca.pem -ca-key=ca-key.pem -config=ca-config.
json -profile=kubernetes admin-csr.json | cfssljson -bare admin
$ ls admin*
admin.csr  admin-csr.json  admin-key.pem  admin.pem
```

创建 kube-proxy 证书

创建 kube-proxy 证书签名请求文件 `kube-proxy-csr.json` :

```
{
  "CN": "system:kube-proxy",
  "hosts": [],
  "key": {
    "algo": "rsa",
    "size": 2048
  },
  "names": [
    {
      "C": "CN",
      "ST": "BeiJing",
      "L": "BeiJing",
      "O": "k8s",
      "OU": "System"
    }
  ]
}
```

- CN 指定该证书的 User 为 `system:kube-proxy` ;
- `kube-apiserver` 预定义的 RoleBinding `cluster-admin` 将User `system:kube-proxy` 与 Role `system:node-proxier` 绑定，该 Role 授予了调用 `kube-apiserver` Proxy 相关 API 的权限；

生成 kube-proxy 客户端证书和私钥

```
$ cfssl gencert -ca=ca.pem -ca-key=ca-key.pem -config=ca-config.
json -profile=kubernetes  kube-proxy-csr.json | cfssljson -bare
kube-proxy
$ ls kube-proxy*
kube-proxy.csr  kube-proxy-csr.json  kube-proxy-key.pem  kube-pr
oxy.pem
```

校验证书

以 kubernetes 证书为例

使用 `opsnssl` 命令

```
$ openssl x509  -noout -text -in  kubernetes.pem
...
    Signature Algorithm: sha256WithRSAEncryption
        Issuer: C=CN, ST=BeiJing, L=BeiJing, O=k8s, OU=System, CN
=Kubernetes
        Validity
            Not Before: Apr  5 05:36:00 2017 GMT
            Not After : Apr  5 05:36:00 2018 GMT
        Subject: C=CN, ST=BeiJing, L=BeiJing, O=k8s, OU=System,
CN=kubernetes
...
        X509v3 extensions:
            X509v3 Key Usage: critical
                Digital Signature, Key Encipherment
            X509v3 Extended Key Usage:
                TLS Web Server Authentication, TLS Web Client Au
thentication
            X509v3 Basic Constraints: critical
                CA:FALSE
            X509v3 Subject Key Identifier:
                DD:52:04:43:10:13:A9:29:24:17:3A:0E:D7:14:DB:36:
F8:6C:E0:E0
            X509v3 Authority Key Identifier:
                keyid:44:04:3B:60:BD:69:78:14:68:AF:A0:41:13:F6:
17:07:13:63:58:CD

            X509v3 Subject Alternative Name:
                DNS:kubernetes, DNS:kubernetes.default, DNS:kube
rnetes.default.svc, DNS:kubernetes.default.svc.cluster, DNS:kube
rnetes.default.svc.cluster.local, IP Address:127.0.0.1, IP Addre
ss:172.20.0.112, IP Address:172.20.0.113, IP Address:172.20.0.11
4, IP Address:172.20.0.115, IP Address:10.254.0.1
...
```

- 确认 `Issuer` 字段的内容和 `ca-csr.json` 一致；
- 确认 `Subject` 字段的内容和 `kubernetes-csr.json` 一致；
- 确认 `X509v3 Subject Alternative Name` 字段的内容和 `kubernetes-csr.json` 一致；
- 确认 `X509v3 Key Usage、Extended Key Usage` 字段的内容和 `ca-config.json` 中 `kubernetes` profile 一致；

使用 `cfssl-certinfo` 命令

```
$ cfssl-certinfo -cert kubernetes.pem
...
{
  "subject": {
    "common_name": "kubernetes",
    "country": "CN",
    "organization": "k8s",
    "organizational_unit": "System",
    "locality": "BeiJing",
    "province": "BeiJing",
    "names": [
      "CN",
      "BeiJing",
      "BeiJing",
      "k8s",
      "System",
      "kubernetes"
    ]
  },
  "issuer": {
    "common_name": "Kubernetes",
    "country": "CN",
    "organization": "k8s",
    "organizational_unit": "System",
    "locality": "BeiJing",
    "province": "BeiJing",
    "names": [
      "CN",
      "BeiJing",
      "BeiJing",
      "k8s",
      "System",
      "Kubernetes"
    ]
  },
  "serial_number": "174360492872423263473151971632292895707129022309",
  "sans": [
    "kubernetes",
    "kubernetes.default",
    "kubernetes.default.svc",
    "kubernetes.default.svc.cluster",
    "kubernetes.default.svc.cluster.local",
    "127.0.0.1",
    "10.64.3.7",
    "10.254.0.1"
  ],
  "not_before": "2017-04-05T05:36:00Z",
  "not_after": "2018-04-05T05:36:00Z",
  "sigalg": "SHA256WithRSA",
...
```

分发证书

将生成的证书和秘钥文件（后缀名为 `.pem` ）拷贝到所有机器的 `/etc/kubernetes/ssl` 目录下备用；

```
mkdir -p /etc/kubernetes/ssl
cp *.pem /etc/kubernetes/ssl
```

参考

- Generate self-signed certificates
- Setting up a Certificate Authority and Creating TLS Certificates
- Client Certificates V/s Server Certificates
- 数字证书及 CA 的扫盲介绍
- TLS bootstrap 引导程序

创建 kubeconfig 文件

注意：请先参考 安装kubectl命令行工具，先在 master 节点上安装 kubectl 然后再进行下面的操作。

`kubelet` 、 `kube-proxy` 等 Node 机器上的进程与 Master 机器的 `kube-apiserver` 进程通信时需要认证和授权；

kubernetes 1.4 开始支持由 `kube-apiserver` 为客户端生成 TLS 证书的 TLS Bootstrapping 功能，这样就不需要为每个客户端生成证书了；该功能当前仅支持为 **`kubelet`** 生成证书；

因为我的master节点和node节点复用，所有在这一步其实已经安装了kubectl。参考安装kubectl命令行工具。

以下操作只需要在master节点上执行，生成的 `*.kubeconfig` 文件可以直接拷贝到node节点的 `/etc/kubernetes` 目录下。

创建 TLS Bootstrapping Token

Token auth file

Token可以是任意的包含128 bit的字符串，可以使用安全的随机数发生器生成。

```
export BOOTSTRAP_TOKEN=$(head -c 16 /dev/urandom | od -An -t x |
tr -d ' ')
cat > token.csv <<EOF
${BOOTSTRAP_TOKEN},kubelet-bootstrap,10001,"system:kubelet-boots
trap"
EOF
```

> 后三行是一句，直接复制上面的脚本运行即可。

注意：在进行后续操作前请检查 **`token.csv`** 文件，确认其中的 **`${BOOTSTRAP_TOKEN}`** 环境变量已经被真实的值替换。

BOOTSTRAP_TOKEN 将被写入到 kube-apiserver 使用的 token.csv 文件和 kubelet 使用的 `bootstrap.kubeconfig` 文件，如果后续重新生成了 BOOTSTRAP_TOKEN，则需要：

1. 更新 token.csv 文件，分发到所有机器 (master 和 node）的 /etc/kubernetes/ 目录下，分发到node节点上非必需；
2. 重新生成 bootstrap.kubeconfig 文件，分发到所有 node 机器的 /etc/kubernetes/ 目录下；
3. 重启 kube-apiserver 和 kubelet 进程；
4. 重新 approve kubelet 的 csr 请求；

```
cp token.csv /etc/kubernetes/
```

创建 kubelet bootstrapping kubeconfig 文件

执行下面的命令时需要先安装kubectl命令，请参考安装kubectl命令行工具。

```
cd /etc/kubernetes
export KUBE_APISERVER="https://172.20.0.113:6443"

# 设置集群参数
kubectl config set-cluster kubernetes \
  --certificate-authority=/etc/kubernetes/ssl/ca.pem \
  --embed-certs=true \
  --server=${KUBE_APISERVER} \
  --kubeconfig=bootstrap.kubeconfig

# 设置客户端认证参数
kubectl config set-credentials kubelet-bootstrap \
  --token=${BOOTSTRAP_TOKEN} \
  --kubeconfig=bootstrap.kubeconfig

# 设置上下文参数
kubectl config set-context default \
  --cluster=kubernetes \
  --user=kubelet-bootstrap \
  --kubeconfig=bootstrap.kubeconfig

# 设置默认上下文
kubectl config use-context default --kubeconfig=bootstrap.kubeco
nfig
```

- `--embed-certs` 为 `true` 时表示将 `certificate-authority` 证书写入

到生成的 `bootstrap.kubeconfig` 文件中；
- 设置客户端认证参数时没有指定秘钥和证书，后续由 `kube-apiserver` 自动生成；

创建 kube-proxy kubeconfig 文件

```
export KUBE_APISERVER="https://172.20.0.113:6443"
# 设置集群参数
kubectl config set-cluster kubernetes \
  --certificate-authority=/etc/kubernetes/ssl/ca.pem \
  --embed-certs=true \
  --server=${KUBE_APISERVER} \
  --kubeconfig=kube-proxy.kubeconfig
# 设置客户端认证参数
kubectl config set-credentials kube-proxy \
  --client-certificate=/etc/kubernetes/ssl/kube-proxy.pem \
  --client-key=/etc/kubernetes/ssl/kube-proxy-key.pem \
  --embed-certs=true \
  --kubeconfig=kube-proxy.kubeconfig
# 设置上下文参数
kubectl config set-context default \
  --cluster=kubernetes \
  --user=kube-proxy \
  --kubeconfig=kube-proxy.kubeconfig
# 设置默认上下文
kubectl config use-context default --kubeconfig=kube-proxy.kubec
onfig
```

- 设置集群参数和客户端认证参数时 `--embed-certs` 都为 `true`，这会将 `certificate-authority`、`client-certificate` 和 `client-key` 指向的证书文件内容写入到生成的 `kube-proxy.kubeconfig` 文件中；
- `kube-proxy.pem` 证书中 CN 为 `system:kube-proxy`，`kube-apiserver` 预定义的 RoleBinding `cluster-admin` 将User `system:kube-proxy` 与 Role `system:node-proxier` 绑定，该 Role 授予了调用 `kube-apiserver` Proxy 相关 API 的权限；

分发 kubeconfig 文件

将两个 kubeconfig 文件分发到所有 Node 机器的 `/etc/kubernetes/` 目录

```
cp bootstrap.kubeconfig kube-proxy.kubeconfig /etc/kubernetes/
```

参考

关于 kubeconfig 文件的更多信息请参考 使用 kubeconfig 文件配置跨集群认证。

创建高可用 etcd 集群

kuberntes 系统使用 etcd 存储所有数据，本文档介绍部署一个三节点高可用 etcd 集群的步骤，这三个节点复用 kubernetes master 机器，分别命名为 `test-001.jimmysong.io`、`test-002.jimmysong.io`、`test-003.jimmysong.io`：

- test-001.jimmysong.io：172.20.0.113
- test-002.jimmysong.io：172.20.0.114
- test-003.jimmysong.io：172.20.0.115

TLS 认证文件

需要为 etcd 集群创建加密通信的 TLS 证书，这里复用以前创建的 kubernetes 证书

```
cp ca.pem kubernetes-key.pem kubernetes.pem /etc/kubernetes/ssl
```

- kubernetes 证书的 `hosts` 字段列表中包含上面三台机器的 IP，否则后续证书校验会失败；

下载二进制文件

到 `https://github.com/coreos/etcd/releases` 页面下载最新版本的二进制文件

```
wget https://github.com/coreos/etcd/releases/download/v3.1.5/etc
d-v3.1.5-linux-amd64.tar.gz
tar -xvf etcd-v3.1.5-linux-amd64.tar.gz
mv etcd-v3.1.5-linux-amd64/etcd* /usr/local/bin
```

或者直接使用yum命令安装：

```
yum install etcd
```

若使用yum安装，默认etcd命令将在 `/usr/bin` 目录下，注意修改下面 的 `etcd.service` 文件中的启动命令地址为 `/usr/bin/etcd` 。

创建 etcd 的 systemd unit 文件

在/usr/lib/systemd/system/目录下创建文件etcd.service，内容如下。注意替换IP地址为你自己的etcd集群的主机IP。

```
[Unit]
Description=Etcd Server
After=network.target
After=network-online.target
Wants=network-online.target
Documentation=https://github.com/coreos

[Service]
Type=notify
WorkingDirectory=/var/lib/etcd/
EnvironmentFile=-/etc/etcd/etcd.conf
ExecStart=/usr/local/bin/etcd \
  --name ${ETCD_NAME} \
  --cert-file=/etc/kubernetes/ssl/kubernetes.pem \
  --key-file=/etc/kubernetes/ssl/kubernetes-key.pem \
  --peer-cert-file=/etc/kubernetes/ssl/kubernetes.pem \
  --peer-key-file=/etc/kubernetes/ssl/kubernetes-key.pem \
  --trusted-ca-file=/etc/kubernetes/ssl/ca.pem \
  --peer-trusted-ca-file=/etc/kubernetes/ssl/ca.pem \
  --initial-advertise-peer-urls ${ETCD_INITIAL_ADVERTISE_PEER_UR
LS} \
  --listen-peer-urls ${ETCD_LISTEN_PEER_URLS} \
  --listen-client-urls ${ETCD_LISTEN_CLIENT_URLS},http://127.0.0
.1:2379 \
  --advertise-client-urls ${ETCD_ADVERTISE_CLIENT_URLS} \
  --initial-cluster-token ${ETCD_INITIAL_CLUSTER_TOKEN} \
  --initial-cluster infra1=https://172.20.0.113:2380,infra2=http
s://172.20.0.114:2380,infra3=https://172.20.0.115:2380 \
  --initial-cluster-state new \
  --data-dir=${ETCD_DATA_DIR}
Restart=on-failure
RestartSec=5
LimitNOFILE=65536

[Install]
WantedBy=multi-user.target
```

- 指定 `etcd` 的工作目录为 `/var/lib/etcd` ，数据目录为 `/var/lib/etcd` ，需在启动服务前创建这个目录，否则启动服务的时候会报

错“Failed at step CHDIR spawning /usr/bin/etcd: No such file or directory”；
- 为了保证通信安全，需要指定 etcd 的公私钥(cert-file和key-file)、Peers 通信的公私钥和 CA 证书(peer-cert-file、peer-key-file、peer-trusted-ca-file)、客户端的CA证书（trusted-ca-file）；
- 创建 `kubernetes.pem` 证书时使用的 `kubernetes-csr.json` 文件的 `hosts` 字段包含所有 **etcd** 节点的**IP**，否则证书校验会出错；
- `--initial-cluster-state` 值为 `new` 时，`--name` 的参数值必须位于 `--initial-cluster` 列表中；

完整 unit 文件见：etcd.service

环境变量配置文件 `/etc/etcd/etcd.conf` 。

```
# [member]
ETCD_NAME=infra1
ETCD_DATA_DIR="/var/lib/etcd"
ETCD_LISTEN_PEER_URLS="https://172.20.0.113:2380"
ETCD_LISTEN_CLIENT_URLS="https://172.20.0.113:2379"

#[cluster]
ETCD_INITIAL_ADVERTISE_PEER_URLS="https://172.20.0.113:2380"
ETCD_INITIAL_CLUSTER_TOKEN="etcd-cluster"
ETCD_ADVERTISE_CLIENT_URLS="https://172.20.0.113:2379"
```

这是172.20.0.113节点的配置，其他两个etcd节点只要将上面的IP地址改成相应节点的IP地址即可。ETCD_NAME换成对应节点的infra1/2/3。

启动 etcd 服务

```
mv etcd.service /usr/lib/systemd/system/
systemctl daemon-reload
systemctl enable etcd
systemctl start etcd
systemctl status etcd
```

在所有的 kubernetes master 节点重复上面的步骤，直到所有机器的 etcd 服务都已启动。

验证服务

在任一 kubernetes master 机器上执行如下命令：

```
$ etcdctl \
  --ca-file=/etc/kubernetes/ssl/ca.pem \
  --cert-file=/etc/kubernetes/ssl/kubernetes.pem \
  --key-file=/etc/kubernetes/ssl/kubernetes-key.pem \
  cluster-health
2017-04-11 15:17:09.082250 I | warning: ignoring ServerName for
user-provided CA for backwards compatibility is deprecated
2017-04-11 15:17:09.083681 I | warning: ignoring ServerName for
user-provided CA for backwards compatibility is deprecated
member 9a2ec640d25672e5 is healthy: got healthy result from http
s://172.20.0.115:2379
member bc6f27ae3be34308 is healthy: got healthy result from http
s://172.20.0.114:2379
member e5c92ea26c4edba0 is healthy: got healthy result from http
s://172.20.0.113:2379
cluster is healthy
```

结果最后一行为 `cluster is healthy` 时表示集群服务正常。

更多资料

关于如何在etcd中查看kubernetes的数据，请参考使用etcdctl访问kuberentes数据。

安装kubectl命令行工具

本文档介绍下载和配置 kubernetes 集群命令行工具 kubelet 的步骤。

下载 kubectl

注意请下载对应的Kubernetes版本的安装包。

```
wget https://dl.k8s.io/v1.6.0/kubernetes-client-linux-amd64.tar.
gz
tar -xzvf kubernetes-client-linux-amd64.tar.gz
cp kubernetes/client/bin/kube* /usr/bin/
chmod a+x /usr/bin/kube*
```

创建 kubectl kubeconfig 文件

```
export KUBE_APISERVER="https://172.20.0.113:6443"
# 设置集群参数
kubectl config set-cluster kubernetes \
  --certificate-authority=/etc/kubernetes/ssl/ca.pem \
  --embed-certs=true \
  --server=${KUBE_APISERVER}
# 设置客户端认证参数
kubectl config set-credentials admin \
  --client-certificate=/etc/kubernetes/ssl/admin.pem \
  --embed-certs=true \
  --client-key=/etc/kubernetes/ssl/admin-key.pem
# 设置上下文参数
kubectl config set-context kubernetes \
  --cluster=kubernetes \
  --user=admin
# 设置默认上下文
kubectl config use-context kubernetes
```

- `admin.pem` 证书 OU 字段值为 `system:masters` ，`kube-apiserver` 预定义的 RoleBinding `cluster-admin` 将 Group `system:masters` 与 Role `cluster-admin` 绑定，该 Role 授予了调用 `kube-apiserver` 相关 API 的权限；
- 生成的 kubeconfig 被保存到 `~/.kube/config` 文件；

注意：`~/.kube/config` 文件拥有对该集群的最高权限，请妥善保管。

更多资料

- kubectl命令概览
- kubectl命令技巧大全

部署master节点

kubernetes master 节点包含的组件：

- kube-apiserver
- kube-scheduler
- kube-controller-manager

目前这三个组件需要部署在同一台机器上。

- `kube-scheduler`、`kube-controller-manager` 和 `kube-apiserver` 三者的功能紧密相关；
- 同时只能有一个 `kube-scheduler`、`kube-controller-manager` 进程处于工作状态，如果运行多个，则需要通过选举产生一个 leader；

注：

- 暂时未实现master节点的高可用
- master节点上没有部署flannel网络插件，如果想要在master节点上也能访问ClusterIP，请参考下一节部署node节点中的配置Flanneld部分。

TLS 证书文件

以下 `pem` 证书文件我们在创建TLS证书和秘钥这一步中已经创建过了，`token.csv` 文件在创建kubeconfig文件的时候创建。我们再检查一下。

```
$ ls /etc/kubernetes/ssl
admin-key.pem  admin.pem  ca-key.pem  ca.pem  kube-proxy-key.pem
  kube-proxy.pem  kubernetes-key.pem  kubernetes.pem
```

下载最新版本的二进制文件

有两种下载方式，请注意下载对应的Kubernetes版本。

方式一

从 github release 页面 下载发布版 tarball，解压后再执行下载脚本

```
wget https://github.com/kubernetes/kubernetes/releases/download/v1.6.0/kubernetes.tar.gz
tar -xzvf kubernetes.tar.gz
cd kubernetes
./cluster/get-kube-binaries.sh
```

方式二

从 `CHANGELOG`页面 下载 `client` 或 `server` tarball 文件

`server` 的 tarball `kubernetes-server-linux-amd64.tar.gz` 已经包含了 `client`(`kubectl`)二进制文件，所以不用单独下载 `kubernetes-client-linux-amd64.tar.gz` 文件；

```
# wget https://dl.k8s.io/v1.6.0/kubernetes-client-linux-amd64.tar.gz
wget https://dl.k8s.io/v1.6.0/kubernetes-server-linux-amd64.tar.gz
tar -xzvf kubernetes-server-linux-amd64.tar.gz
cd kubernetes
tar -xzvf  kubernetes-src.tar.gz
```

将二进制文件拷贝到指定路径

```
cp -r server/bin/{kube-apiserver,kube-controller-manager,kube-scheduler,kubectl,kube-proxy,kubelet} /usr/local/bin/
```

配置和启动 kube-apiserver

创建 **kube-apiserver的service配置文件**

service配置文件 `/usr/lib/systemd/system/kube-apiserver.service` 内容：

```
[Unit]
Description=Kubernetes API Service
Documentation=https://github.com/GoogleCloudPlatform/kubernetes
After=network.target
After=etcd.service

[Service]
EnvironmentFile=-/etc/kubernetes/config
EnvironmentFile=-/etc/kubernetes/apiserver
ExecStart=/usr/local/bin/kube-apiserver \
        $KUBE_LOGTOSTDERR \
        $KUBE_LOG_LEVEL \
        $KUBE_ETCD_SERVERS \
        $KUBE_API_ADDRESS \
        $KUBE_API_PORT \
        $KUBELET_PORT \
        $KUBE_ALLOW_PRIV \
        $KUBE_SERVICE_ADDRESSES \
        $KUBE_ADMISSION_CONTROL \
        $KUBE_API_ARGS
Restart=on-failure
Type=notify
LimitNOFILE=65536

[Install]
WantedBy=multi-user.target
```

`/etc/kubernetes/config` 文件的内容为：

```
###
# kubernetes system config
#
# The following values are used to configure various aspects of
all
# kubernetes services, including
#
#   kube-apiserver.service
#   kube-controller-manager.service
#   kube-scheduler.service
#   kubelet.service
#   kube-proxy.service
# logging to stderr means we get it in the systemd journal
KUBE_LOGTOSTDERR="--logtostderr=true"

# journal message level, 0 is debug
KUBE_LOG_LEVEL="--v=0"

# Should this cluster be allowed to run privileged docker contai
ners
KUBE_ALLOW_PRIV="--allow-privileged=true"

# How the controller-manager, scheduler, and proxy find the apis
erver
#KUBE_MASTER="--master=http://test-001.jimmysong.io:8080"
KUBE_MASTER="--master=http://172.20.0.113:8080"
```

该配置文件同时被kube-apiserver、kube-controller-manager、kube-scheduler、kubelet、kube-proxy使用。

apiserver配置文件 `/etc/kubernetes/apiserver` 内容为：

```
###
## kubernetes system config
##
## The following values are used to configure the kube-apiserver
##
#
## The address on the local server to listen to.
#KUBE_API_ADDRESS="--insecure-bind-address=test-001.jimmysong.io
"
KUBE_API_ADDRESS="--advertise-address=172.20.0.113 --bind-addres
s=172.20.0.113 --insecure-bind-address=172.20.0.113"
#
## The port on the local server to listen on.
#KUBE_API_PORT="--port=8080"
#
## Port minions listen on
#KUBELET_PORT="--kubelet-port=10250"
#
## Comma separated list of nodes in the etcd cluster
KUBE_ETCD_SERVERS="--etcd-servers=https://172.20.0.113:2379,http
s://172.20.0.114:2379,https://172.20.0.115:2379"
#
## Address range to use for services
KUBE_SERVICE_ADDRESSES="--service-cluster-ip-range=10.254.0.0/16
"
#
## default admission control policies
KUBE_ADMISSION_CONTROL="--admission-control=ServiceAccount,Names
paceLifecycle,NamespaceExists,LimitRanger,ResourceQuota"
#
## Add your own!
KUBE_API_ARGS="--authorization-mode=RBAC --runtime-config=rbac.a
uthorization.k8s.io/v1beta1 --kubelet-https=true --experimental-
bootstrap-token-auth --token-auth-file=/etc/kubernetes/token.csv
 --service-node-port-range=30000-32767 --tls-cert-file=/etc/kube
rnetes/ssl/kubernetes.pem --tls-private-key-file=/etc/kubernetes
/ssl/kubernetes-key.pem --client-ca-file=/etc/kubernetes/ssl/ca.
pem --service-account-key-file=/etc/kubernetes/ssl/ca-key.pem --
etcd-cafile=/etc/kubernetes/ssl/ca.pem --etcd-certfile=/etc/kube
rnetes/ssl/kubernetes.pem --etcd-keyfile=/etc/kubernetes/ssl/kub
ernetes-key.pem --enable-swagger-ui=true --apiserver-count=3 --a
udit-log-maxage=30 --audit-log-maxbackup=3 --audit-log-maxsize=1
00 --audit-log-path=/var/lib/audit.log --event-ttl=1h"
```

- `--experimental-bootstrap-token-auth` Bootstrap Token Authentication 在1.9版本已经变成了正式feature，参数名称改为 `--enable-bootstrap-token-auth`
- 如果中途修改过 `--service-cluster-ip-range` 地址，则必须将default命名空间的 `kubernetes` 的service给删除，使用命令： `kubectl delete`

`service kubernetes` ，然后系统会自动用新的ip重建这个service，不然apiserver的log有报错 `the cluster IP x.x.x.x for service kubernetes/default is not within the service CIDR x.x.x.x/16; please recreate`

- `--authorization-mode=RBAC` 指定在安全端口使用 RBAC 授权模式，拒绝未通过授权的请求；
- kube-scheduler、kube-controller-manager 一般和 kube-apiserver 部署在同一台机器上，它们使用非安全端口和 kube-apiserver通信;
- kubelet、kube-proxy、kubectl 部署在其它 Node 节点上，如果通过安全端口访问 kube-apiserver，则必须先通过 TLS 证书认证，再通过 RBAC 授权；
- kube-proxy、kubectl 通过在使用的证书里指定相关的 User、Group 来达到通过 RBAC 授权的目的；
- 如果使用了 kubelet TLS Boostrap 机制，则不能再指定 `--kubelet-certificate-authority` 、 `--kubelet-client-certificate` 和 `--kubelet-client-key` 选项，否则后续 kube-apiserver 校验 kubelet 证书时出现 ”x509: certificate signed by unknown authority“ 错误；
- `--admission-control` 值必须包含 `ServiceAccount` ；
- `--bind-address` 不能为 `127.0.0.1` ；
- `runtime-config` 配置为 `rbac.authorization.k8s.io/v1beta1` ，表示运行时的apiVersion；
- `--service-cluster-ip-range` 指定 Service Cluster IP 地址段，该地址段不能路由可达；
- 缺省情况下 kubernetes 对象保存在 etcd `/registry` 路径下，可以通过 `--etcd-prefix` 参数进行调整；
- 如果需要开通http的无认证的接口，则可以增加以下两个参数： `--insecure-port=8080 --insecure-bind-address=127.0.0.1` 。注意，生产上不要绑定到非127.0.0.1的地址上

Kubernetes 1.9

- 对于Kubernetes1.9集群，需要注意配置 `KUBE_API_ARGS` 环境变量中的 `--authorization-mode=Node,RBAC` ，增加对 `Node` 授权的模式，否则将无法注册node。
- `--experimental-bootstrap-token-auth` Bootstrap Token Authentication 在kubernetes 1.9版本已经废弃，参数名称改为 `--enable-bootstrap-token-auth`

完整 unit 见 kube-apiserver.service

启动**kube-apiserver**

```
systemctl daemon-reload
systemctl enable kube-apiserver
systemctl start kube-apiserver
systemctl status kube-apiserver
```

配置和启动 kube-controller-manager

创建 **kube-controller-manager**的**serivce**配置文件

文件路径 `/usr/lib/systemd/system/kube-controller-manager.service`

```
[Unit]
Description=Kubernetes Controller Manager
Documentation=https://github.com/GoogleCloudPlatform/kubernetes

[Service]
EnvironmentFile=-/etc/kubernetes/config
EnvironmentFile=-/etc/kubernetes/controller-manager
ExecStart=/usr/local/bin/kube-controller-manager \
        $KUBE_LOGTOSTDERR \
        $KUBE_LOG_LEVEL \
        $KUBE_MASTER \
        $KUBE_CONTROLLER_MANAGER_ARGS
Restart=on-failure
LimitNOFILE=65536

[Install]
WantedBy=multi-user.target
```

配置文件 `/etc/kubernetes/controller-manager` 。

```
###
# The following values are used to configure the kubernetes cont
roller-manager

# defaults from config and apiserver should be adequate

# Add your own!
KUBE_CONTROLLER_MANAGER_ARGS="--address=127.0.0.1 --service-clus
ter-ip-range=10.254.0.0/16 --cluster-name=kubernetes --cluster-s
igning-cert-file=/etc/kubernetes/ssl/ca.pem --cluster-signing-ke
y-file=/etc/kubernetes/ssl/ca-key.pem  --service-account-private
-key-file=/etc/kubernetes/ssl/ca-key.pem --root-ca-file=/etc/kub
ernetes/ssl/ca.pem --leader-elect=true"
```

- `--service-cluster-ip-range` 参数指定 Cluster 中 Service 的CIDR范围，该网络在各 Node 间必须路由不可达，必须和 kube-apiserver 中的参数一致；
- `--cluster-signing-*` 指定的证书和私钥文件用来签名为 TLS BootStrap 创建的证书和私钥；
- `--root-ca-file` 用来对 kube-apiserver 证书进行校验，指定该参数后，才会在**Pod** 容器的 **ServiceAccount** 中放置该 **CA** 证书文件；
- `--address` 值必须为 `127.0.0.1` ，kube-apiserver 期望 scheduler 和 controller-manager 在同一台机器；

启动 kube-controller-manager

```
systemctl daemon-reload
systemctl enable kube-controller-manager
systemctl start kube-controller-manager
systemctl status kube-controller-manager
```

我们启动每个组件后可以通过执行命令 `kubectl get componentstatuses` ，来查看各个组件的状态;

```
$ kubectl get componentstatuses
NAME                 STATUS      MESSAGE

ERROR
scheduler            Unhealthy   Get http://127.0.0.1:10251/heal
thz: dial tcp 127.0.0.1:10251: getsockopt: connection refused
controller-manager   Healthy     ok

etcd-2               Healthy     {"health": "true"}
etcd-0               Healthy     {"health": "true"}

etcd-1               Healthy     {"health": "true"}
```

- 如果有组件report unhealthy请参考：https://github.com/kubernetes-incubator/bootkube/issues/64

完整 unit 见 kube-controller-manager.service

配置和启动 kube-scheduler

创建 **kube-scheduler**的**serivce**配置文件

文件路径 `/usr/lib/systemd/system/kube-scheduler.service` 。

```
[Unit]
Description=Kubernetes Scheduler Plugin
Documentation=https://github.com/GoogleCloudPlatform/kubernetes

[Service]
EnvironmentFile=-/etc/kubernetes/config
EnvironmentFile=-/etc/kubernetes/scheduler
ExecStart=/usr/local/bin/kube-scheduler \
            $KUBE_LOGTOSTDERR \
            $KUBE_LOG_LEVEL \
            $KUBE_MASTER \
            $KUBE_SCHEDULER_ARGS
Restart=on-failure
LimitNOFILE=65536

[Install]
WantedBy=multi-user.target
```

配置文件 `/etc/kubernetes/scheduler` 。

```
###
# kubernetes scheduler config

# default config should be adequate

# Add your own!
KUBE_SCHEDULER_ARGS="--leader-elect=true --address=127.0.0.1"
```

- `--address` 值必须为 `127.0.0.1`，因为当前 kube-apiserver 期望 scheduler 和 controller-manager 在同一台机器；

完整 unit 见 kube-scheduler.service

启动 kube-scheduler

```
systemctl daemon-reload
systemctl enable kube-scheduler
systemctl start kube-scheduler
systemctl status kube-scheduler
```

验证 master 节点功能

```
$ kubectl get componentstatuses
NAME                 STATUS    MESSAGE              ERROR
scheduler            Healthy   ok
controller-manager   Healthy   ok
etcd-0               Healthy   {"health": "true"}
etcd-1               Healthy   {"health": "true"}
etcd-2               Healthy   {"health": "true"}
```

安装flannel网络插件

所有的node节点都需要安装网络插件才能让所有的Pod加入到同一个局域网中，本文是安装flannel网络插件的参考文档。

建议直接使用yum安装flanneld，除非对版本有特殊需求，默认安装的是0.7.1版本的flannel。

```
yum install -y flannel
```

service配置文件 `/usr/lib/systemd/system/flanneld.service` 。

```
[Unit]
Description=Flanneld overlay address etcd agent
After=network.target
After=network-online.target
Wants=network-online.target
After=etcd.service
Before=docker.service

[Service]
Type=notify
EnvironmentFile=/etc/sysconfig/flanneld
EnvironmentFile=-/etc/sysconfig/docker-network
ExecStart=/usr/bin/flanneld-start \
  -etcd-endpoints=${FLANNEL_ETCD_ENDPOINTS} \
  -etcd-prefix=${FLANNEL_ETCD_PREFIX} \
  $FLANNEL_OPTIONS
ExecStartPost=/usr/libexec/flannel/mk-docker-opts.sh -k DOCKER_N
ETWORK_OPTIONS -d /run/flannel/docker
Restart=on-failure

[Install]
WantedBy=multi-user.target
RequiredBy=docker.service
```

`/etc/sysconfig/flanneld` 配置文件：

```
# Flanneld configuration options

# etcd url location.  Point this to the server where etcd runs
FLANNEL_ETCD_ENDPOINTS="https://172.20.0.113:2379,https://172.20.0.114:2379,https://172.20.0.115:2379"

# etcd config key.  This is the configuration key that flannel queries
# For address range assignment
FLANNEL_ETCD_PREFIX="/kube-centos/network"

# Any additional options that you want to pass
FLANNEL_OPTIONS="-etcd-cafile=/etc/kubernetes/ssl/ca.pem -etcd-certfile=/etc/kubernetes/ssl/kubernetes.pem -etcd-keyfile=/etc/kubernetes/ssl/kubernetes-key.pem"
```

如果是多网卡（例如vagrant环境），则需要在FLANNEL_OPTIONS中增加指定的外网出口的网卡，例如-iface=eth2

在etcd中创建网络配置

执行下面的命令为docker分配IP地址段。

```
etcdctl --endpoints=https://172.20.0.113:2379,https://172.20.0.114:2379,https://172.20.0.115:2379 \
  --ca-file=/etc/kubernetes/ssl/ca.pem \
  --cert-file=/etc/kubernetes/ssl/kubernetes.pem \
  --key-file=/etc/kubernetes/ssl/kubernetes-key.pem \
  mkdir /kube-centos/network
etcdctl --endpoints=https://172.20.0.113:2379,https://172.20.0.114:2379,https://172.20.0.115:2379 \
  --ca-file=/etc/kubernetes/ssl/ca.pem \
  --cert-file=/etc/kubernetes/ssl/kubernetes.pem \
  --key-file=/etc/kubernetes/ssl/kubernetes-key.pem \
  mk /kube-centos/network/config '{"Network":"172.30.0.0/16","SubnetLen":24,"Backend":{"Type":"vxlan"}}'
```

如果你要使用 `host-gw` 模式，可以直接将vxlan改成 `host-gw` 即可。

注：参考网络和集群性能测试那节，最终我们使用的 `host-gw` 模式，关于flannel支持的backend模式见：https://github.com/coreos/flannel/blob/master/Documentation/backends.md。

启动flannel

```
systemctl daemon-reload
systemctl enable flanneld
systemctl start flanneld
systemctl status flanneld
```

现在查询etcd中的内容可以看到：

```
$etcdctl --endpoints=${ETCD_ENDPOINTS} \
  --ca-file=/etc/kubernetes/ssl/ca.pem \
  --cert-file=/etc/kubernetes/ssl/kubernetes.pem \
  --key-file=/etc/kubernetes/ssl/kubernetes-key.pem \
  ls /kube-centos/network/subnets
/kube-centos/network/subnets/172.30.14.0-24
/kube-centos/network/subnets/172.30.38.0-24
/kube-centos/network/subnets/172.30.46.0-24

$etcdctl --endpoints=${ETCD_ENDPOINTS} \
  --ca-file=/etc/kubernetes/ssl/ca.pem \
  --cert-file=/etc/kubernetes/ssl/kubernetes.pem \
  --key-file=/etc/kubernetes/ssl/kubernetes-key.pem \
  get /kube-centos/network/config
{ "Network": "172.30.0.0/16", "SubnetLen": 24, "Backend": { "Typ
e": "vxlan" } }

$etcdctl --endpoints=${ETCD_ENDPOINTS} \
  --ca-file=/etc/kubernetes/ssl/ca.pem \
  --cert-file=/etc/kubernetes/ssl/kubernetes.pem \
  --key-file=/etc/kubernetes/ssl/kubernetes-key.pem \
  get /kube-centos/network/subnets/172.30.14.0-24
{"PublicIP":"172.20.0.114","BackendType":"vxlan","BackendData":{
"VtepMAC":"56:27:7d:1c:08:22"}}

$etcdctl --endpoints=${ETCD_ENDPOINTS} \
  --ca-file=/etc/kubernetes/ssl/ca.pem \
  --cert-file=/etc/kubernetes/ssl/kubernetes.pem \
  --key-file=/etc/kubernetes/ssl/kubernetes-key.pem \
  get /kube-centos/network/subnets/172.30.38.0-24
{"PublicIP":"172.20.0.115","BackendType":"vxlan","BackendData":{
"VtepMAC":"12:82:83:59:cf:b8"}}

$etcdctl --endpoints=${ETCD_ENDPOINTS} \
  --ca-file=/etc/kubernetes/ssl/ca.pem \
  --cert-file=/etc/kubernetes/ssl/kubernetes.pem \
  --key-file=/etc/kubernetes/ssl/kubernetes-key.pem \
  get /kube-centos/network/subnets/172.30.46.0-24
{"PublicIP":"172.20.0.113","BackendType":"vxlan","BackendData":{
"VtepMAC":"e6:b2:fd:f6:66:96"}}
```

如果可以查看到以上内容证明flannel已经安装完成，下一步是在node节点上安装和配置docker、kubelet、kube-proxy等，请参考下一节部署node节点。

部署node节点

Kubernetes node节点包含如下组件：

- Flanneld：参考我之前写的文章Kubernetes基于Flannel的网络配置，之前没有配置TLS，现在需要在service配置文件中增加TLS配置，安装过程请参考上一节安装flannel网络插件。
- Docker1.12.5：docker的安装很简单，这里也不说了，但是需要注意docker的配置。
- kubelet：直接用二进制文件安装
- kube-proxy：直接用二进制文件安装

注意：每台 node 上都需要安装 flannel，master 节点上可以不安装。

步骤简介

1. 确认在上一步中我们安装配置的网络插件flannel已启动且运行正常
2. 安装配置docker后启动
3. 安装配置kubelet、kube-proxy后启动
4. 验证

目录和文件

我们再检查一下三个节点上，经过前几步操作我们已经创建了如下的证书和配置文件。

```
$ ls /etc/kubernetes/ssl
admin-key.pem  admin.pem  ca-key.pem  ca.pem  kube-proxy-key.pem
  kube-proxy.pem  kubernetes-key.pem  kubernetes.pem
$ ls /etc/kubernetes/
apiserver  bootstrap.kubeconfig  config  controller-manager  kub
elet  kube-proxy.kubeconfig  proxy  scheduler  ssl  token.csv
```

配置Docker

> 如果您使用yum的方式安装的flannel则不需要执行mk-docker-opts.sh文件这一步，参考Flannel官方文档中的Docker Integration。

如果你不是使用yum安装的flannel，那么需要下载flannel github release中的tar包，解压后会获得一个**mk-docker-opts.sh**文件，到flannel release页面下载对应版本的安装包，该脚本见mk-docker-opts.sh，因为我们使用yum安装所以不需要执行这一步。

这个文件是用来 `Generate Docker daemon options based on flannel env file` 。

使用 `systemctl` 命令启动flanneld后，会自动执行 `./mk-docker-opts.sh -i` 生成如下两个文件环境变量文件：

- /run/flannel/subnet.env

```
FLANNEL_NETWORK=172.30.0.0/16
FLANNEL_SUBNET=172.30.46.1/24
FLANNEL_MTU=1450
FLANNEL_IPMASQ=false
```

- /run/docker_opts.env

```
DOCKER_OPT_BIP="--bip=172.30.46.1/24"
DOCKER_OPT_IPMASQ="--ip-masq=true"
DOCKER_OPT_MTU="--mtu=1450"
```

Docker将会读取这两个环境变量文件作为容器启动参数。

注意：不论您用什么方式安装的flannel，下面这一步是必不可少的。

yum方式安装的flannel

修改docker的配置文件 `/usr/lib/systemd/system/docker.service` ，增加一条环境变量配置：

```
EnvironmentFile=-/run/flannel/docker
```

`/run/flannel/docker` 文件是flannel启动后自动生成的，其中包含了docker启动时需要的参数。

二进制方式安装的flannel

修改docker的配置文件 `/usr/lib/systemd/system/docker.service` ，增加如下几条环境变量配置：

```
EnvironmentFile=-/run/docker_opts.env
EnvironmentFile=-/run/flannel/subnet.env
```

这两个文件是 `mk-docker-opts.sh` 脚本生成环境变量文件默认的保存位置，docker启动的时候需要加载这几个配置文件才可以加入到flannel创建的虚拟网络里。

所以不论您使用何种方式安装的flannel，将以下配置加入到 `docker.service` 中可确保万无一失。

```
EnvironmentFile=-/run/flannel/docker
EnvironmentFile=-/run/docker_opts.env
EnvironmentFile=-/run/flannel/subnet.env
EnvironmentFile=-/etc/sysconfig/docker
EnvironmentFile=-/etc/sysconfig/docker-storage
EnvironmentFile=-/etc/sysconfig/docker-network
EnvironmentFile=-/run/docker_opts.env
```

请参考docker.service中的配置。

启动docker

重启了docker后还要重启kubelet，这时又遇到问题，kubelet启动失败。报错：

```
Mar 31 16:44:41 test-002.jimmysong.io kubelet[81047]: error: fai
led to run Kubelet: failed to create kubelet: misconfiguration:
kubelet cgroup driver: "cgroupfs" is different from docker cgrou
p driver: "systemd"
```

这是kubelet与docker的**cgroup driver**不一致导致的，kubelet启动的时候有个 `–cgroup-driver` 参数可以指定为"cgroupfs"或者“systemd”。

```
--cgroup-driver string                                    Driver
 that the kubelet uses to manipulate cgroups on the host.  Possi
ble values: 'cgroupfs', 'systemd' (default "cgroupfs")
```

配置docker的service配置文件 `/usr/lib/systemd/system/docker.service` ，设置 `ExecStart` 中的 `--exec-opt native.cgroupdriver=systemd` 。

安装和配置kubelet

kubernets1.8

相对于kubernetes1.6集群必须进行的配置有：

对于kuberentes1.8集群，必须关闭swap，否则kubelet启动将失败。

修改 `/etc/fstab` 将，swap系统注释掉。

---

kubelet 启动时向 kube-apiserver 发送 TLS bootstrapping 请求，需要先将 bootstrap token 文件中的 kubelet-bootstrap 用户赋予 system:node-bootstrapper cluster 角色(role)， 然后 kubelet 才能有权限创建认证请求(certificate signing requests)：

```
cd /etc/kubernetes
kubectl create clusterrolebinding kubelet-bootstrap \
  --clusterrole=system:node-bootstrapper \
  --user=kubelet-bootstrap
```

- `--user=kubelet-bootstrap` 是在 `/etc/kubernetes/token.csv` 文件中指定的用户名，同时也写入了 `/etc/kubernetes/bootstrap.kubeconfig` 文件；

下载最新的kubelet和kube-proxy二进制文件

注意请下载对应的Kubernetes版本的安装包。

```
wget https://dl.k8s.io/v1.6.0/kubernetes-server-linux-amd64.tar.
gz
tar -xzvf kubernetes-server-linux-amd64.tar.gz
cd kubernetes
tar -xzvf  kubernetes-src.tar.gz
cp -r ./server/bin/{kube-proxy,kubelet} /usr/local/bin/
```

创建kubelet的service配置文件

文件位置 `/usr/lib/systemd/system/kubelet.service` 。

```
[Unit]
Description=Kubernetes Kubelet Server
Documentation=https://github.com/GoogleCloudPlatform/kubernetes
After=docker.service
Requires=docker.service

[Service]
WorkingDirectory=/var/lib/kubelet
EnvironmentFile=-/etc/kubernetes/config
EnvironmentFile=-/etc/kubernetes/kubelet
ExecStart=/usr/local/bin/kubelet \
            $KUBE_LOGTOSTDERR \
            $KUBE_LOG_LEVEL \
            $KUBELET_API_SERVER \
            $KUBELET_ADDRESS \
            $KUBELET_PORT \
            $KUBELET_HOSTNAME \
            $KUBE_ALLOW_PRIV \
            $KUBELET_POD_INFRA_CONTAINER \
            $KUBELET_ARGS
Restart=on-failure

[Install]
WantedBy=multi-user.target
```

kubelet的配置文件 `/etc/kubernetes/kubelet` 。其中的IP地址更改为你的每台node节点的IP地址。

注意：在启动kubelet之前，需要先手动创建 `/var/lib/kubelet` 目录。

下面是kubelet的配置文件 `/etc/kubernetes/kubelet` :

kubernetes1.8

相对于kubenrete1.6的配置变动：

- 对于kuberentes1.8集群中的kubelet配置，取消了 `KUBELET_API_SERVER` 的配置，而改用kubeconfig文件来定义master地址，所以请注释掉 `KUBELET_API_SERVER` 配置。

```
###
## kubernetes kubelet (minion) config
#
## The address for the info server to serve on (set to 0.0.0.0 o
r "" for all interfaces)
KUBELET_ADDRESS="--address=172.20.0.113"
#
## The port for the info server to serve on
#KUBELET_PORT="--port=10250"
#
## You may leave this blank to use the actual hostname
KUBELET_HOSTNAME="--hostname-override=172.20.0.113"
#
## location of the api-server
## COMMENT THIS ON KUBERNETES 1.8+
KUBELET_API_SERVER="--api-servers=http://172.20.0.113:8080"
#
## pod infrastructure container
KUBELET_POD_INFRA_CONTAINER="--pod-infra-container-image=harbor-
001.jimmysong.io/library/pod-infrastructure:rhel7"
#
## Add your own!
KUBELET_ARGS="--cgroup-driver=systemd --cluster-dns=10.254.0.2 -
-experimental-bootstrap-kubeconfig=/etc/kubernetes/bootstrap.kub
econfig --kubeconfig=/etc/kubernetes/kubelet.kubeconfig --requir
e-kubeconfig --cert-dir=/etc/kubernetes/ssl --cluster-domain=clu
ster.local --hairpin-mode promiscuous-bridge --serialize-image-p
ulls=false"
```

- 如果使用systemd方式启动，则需要额外增加两个参数 `--runtime-cgroups=/systemd/system.slice --kubelet-cgroups=/systemd/system.slice`
- `--experimental-bootstrap-kubeconfig` 在1.9版本已经变成了 `--bootstrap-kubeconfig`
- `--address` 不能设置为 `127.0.0.1`，否则后续 Pods 访问 kubelet 的 API 接口时会失败，因为 Pods 访问的 `127.0.0.1` 指向自己而不是 kubelet；
- 如果设置了 `--hostname-override` 选项，则 `kube-proxy` 也需要设置该选项，否则会出现找不到 Node 的情况；
- `"--cgroup-driver` 配置成 `systemd`，不要使用 `cgroup`，否则在 CentOS 系统中 kubelet 将启动失败（保持docker和kubelet中的cgroup driver配置一致即可，不一定非使用 `systemd`）。
- `--experimental-bootstrap-kubeconfig` 指向 bootstrap kubeconfig 文件，kubelet 使用该文件中的用户名和 token 向 kube-apiserver 发送 TLS Bootstrapping 请求；

- 管理员通过了 CSR 请求后，kubelet 自动在 `--cert-dir` 目录创建证书和私钥文件(`kubelet-client.crt` 和 `kubelet-client.key`)，然后写入 `--kubeconfig` 文件；
- 建议在 `--kubeconfig` 配置文件中指定 `kube-apiserver` 地址，如果未指定 `--api-servers` 选项，则必须指定 `--require-kubeconfig` 选项后才从配置文件中读取 kube-apiserver 的地址，否则 kubelet 启动后将找不到 kube-apiserver (日志中提示未找到 API Server），`kubectl get nodes` 不会返回对应的 Node 信息;
- `--cluster-dns` 指定 kubedns 的 Service IP(可以先分配，后续创建 kubedns 服务时指定该 IP)，`--cluster-domain` 指定域名后缀，这两个参数同时指定后才会生效；
- `--cluster-domain` 指定 pod 启动时 `/etc/resolve.conf` 文件中的 `search domain` ，起初我们将其配置成了 `cluster.local.` ，这样在解析 service 的 DNS 名称时是正常的，可是在解析 headless service 中的 FQDN pod name 的时候却错误，因此我们将其修改为 `cluster.local` ，去掉嘴后面的 ”点号“ 就可以解决该问题，关于 kubernetes 中的域名/服务名称解析请参见我的另一篇文章。
- `--kubeconfig=/etc/kubernetes/kubelet.kubeconfig` 中指定的 `kubelet.kubeconfig` 文件在第一次启动kubelet之前并不存在，请看下文，当通过CSR请求后会自动生成 `kubelet.kubeconfig` 文件，如果你的节点上已经生成了 `~/.kube/config` 文件，你可以将该文件拷贝到该路径下，并重命名为 `kubelet.kubeconfig` ，所有node节点可以共用同一个kubelet.kubeconfig文件，这样新添加的节点就不需要再创建CSR请求就能自动添加到kubernetes集群中。同样，在任意能够访问到kubernetes集群的主机上使用 `kubectl --kubeconfig` 命令操作集群时，只要使用 `~/.kube/config` 文件就可以通过权限认证，因为这里面已经有认证信息并认为你是admin用户，对集群拥有所有权限。
- `KUBELET_POD_INFRA_CONTAINER` 是基础镜像容器，这里我用的是私有镜像仓库地址，大家部署的时候需要修改为自己的镜像。我上传了一个到时速云上，可以直接 `docker pull index.tenxcloud.com/jimmy/pod-infrastructure` 下载。`pod-infrastructure` 镜像是Redhat制作的，大小接近80M，下载比较耗时，其实该镜像并不运行什么具体进程，可以使用Google的pause镜像 `gcr.io/google_containers/pause-amd64:3.0` ，这个镜像只有300多K，或者通过DockerHub下载 `jimmysong/pause-amd64:3.0` 。

完整 unit 见 kubelet.service

启动kublet

```
systemctl daemon-reload
systemctl enable kubelet
systemctl start kubelet
systemctl status kubelet
```

通过 kublet 的 TLS 证书请求

kubelet 首次启动时向 kube-apiserver 发送证书签名请求，必须通过后 kubernetes 系统才会将该 Node 加入到集群。

查看未授权的 CSR 请求

```
$ kubectl get csr
NAME          AGE       REQUESTOR           CONDITION
csr-2b308     4m        kubelet-bootstrap   Pending
$ kubectl get nodes
No resources found.
```

通过 CSR 请求

```
$ kubectl certificate approve csr-2b308
certificatesigningrequest "csr-2b308" approved
$ kubectl get nodes
NAME          STATUS     AGE       VERSION
10.64.3.7     Ready      49m       v1.6.1
```

自动生成了 kubelet kubeconfig 文件和公私钥

```
$ ls -l /etc/kubernetes/kubelet.kubeconfig
-rw------- 1 root root 2284 Apr  7 02:07 /etc/kubernetes/kubelet
.kubeconfig
$ ls -l /etc/kubernetes/ssl/kubelet*
-rw-r--r-- 1 root root 1046 Apr  7 02:07 /etc/kubernetes/ssl/kub
elet-client.crt
-rw------- 1 root root  227 Apr  7 02:04 /etc/kubernetes/ssl/kub
elet-client.key
-rw-r--r-- 1 root root 1103 Apr  7 02:07 /etc/kubernetes/ssl/kub
elet.crt
-rw------- 1 root root 1675 Apr  7 02:07 /etc/kubernetes/ssl/kub
elet.key
```

假如你更新kubernetes的证书，只要没有更新 `token.csv` ，当重启kubelet后，该node就会自动加入到kuberentes集群中，而不会重新发送 `certificaterequest` ，也不需要在master节点上执行 `kubectl certificate approve` 操作。前提是不要删除node节点上的 `/etc/kubernetes/ssl/kubelet*` 和 `/etc/kubernetes/kubelet.kubeconfig` 文件。否则kubelet启动时会提示找不到证书而失败。

注意：如果启动kubelet的时候见到证书相关的报错，有个trick可以解决这个问题，可以将master节点上的 `~/.kube/config` 文件（该文件在安装kubectl命令行工具这一步中将会自动生成）拷贝到node节点的 `/etc/kubernetes/kubelet.kubeconfig` 位置，这样就不需要通过CSR，当kubelet启动后就会自动加入的集群中。

配置 kube-proxy

安装conntrack

```
yum install -y conntrack-tools
```

创建 kube-proxy 的service配置文件

文件路径 `/usr/lib/systemd/system/kube-proxy.service` 。

```
[Unit]
Description=Kubernetes Kube-Proxy Server
Documentation=https://github.com/GoogleCloudPlatform/kubernetes
After=network.target

[Service]
EnvironmentFile=-/etc/kubernetes/config
EnvironmentFile=-/etc/kubernetes/proxy
ExecStart=/usr/local/bin/kube-proxy \
        $KUBE_LOGTOSTDERR \
        $KUBE_LOG_LEVEL \
        $KUBE_MASTER \
        $KUBE_PROXY_ARGS
Restart=on-failure
LimitNOFILE=65536

[Install]
WantedBy=multi-user.target
```

kube-proxy配置文件 `/etc/kubernetes/proxy` 。

```
###
# kubernetes proxy config

# default config should be adequate

# Add your own!
KUBE_PROXY_ARGS="--bind-address=172.20.0.113 --hostname-override
=172.20.0.113 --kubeconfig=/etc/kubernetes/kube-proxy.kubeconfig
 --cluster-cidr=10.254.0.0/16"
```

- `--hostname-override` 参数值必须与 kubelet 的值一致，否则 kube-proxy 启动后会找不到该 Node，从而不会创建任何 iptables 规则；
- kube-proxy 根据 `--cluster-cidr` 判断集群内部和外部流量，指定 `--cluster-cidr` 或 `--masquerade-all` 选项后 kube-proxy 才会对访问 Service IP 的请求做 SNAT；
- `--kubeconfig` 指定的配置文件嵌入了 kube-apiserver 的地址、用户名、证书、秘钥等请求和认证信息；
- 预定义的 RoleBinding `cluster-admin` 将User `system:kube-proxy` 与 Role `system:node-proxier` 绑定，该 Role 授予了调用 `kube-apiserver` Proxy 相关 API 的权限；

完整 unit 见 kube-proxy.service

启动 kube-proxy

```
systemctl daemon-reload
systemctl enable kube-proxy
systemctl start kube-proxy
systemctl status kube-proxy
```

验证测试

我们创建一个nginx的service试一下集群是否可用。

```
$ kubectl run nginx --replicas=2 --labels="run=load-balancer-exa
mple" --image=harbor-001.jimmysong.io/library/nginx:1.9  --port=
80
deployment "nginx" created
$ kubectl expose deployment nginx --type=NodePort --name=example
-service
service "example-service" exposed
$ kubectl describe svc example-service
Name:                example-service
Namespace:            default
Labels:                run=load-balancer-example
Annotations:            <none>
Selector:            run=load-balancer-example
Type:                NodePort
IP:                10.254.62.207
Port:                <unset>    80/TCP
NodePort:            <unset>    32724/TCP
Endpoints:            172.30.60.2:80,172.30.94.2:80
Session Affinity:    None
Events:                <none>
$ curl "10.254.62.207:80"
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
    body {
        width: 35em;
        margin: 0 auto;
        font-family: Tahoma, Verdana, Arial, sans-serif;
    }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully in
stalled and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
```

提示：上面的测试示例中使用的nginx是我的私有镜像仓库中的镜像 `harbor-001.jimmysong.io/library/nginx:1.9` ，大家在测试过程中请换成自己的nginx镜像地址。

访问 `172.20.0.113:32724` 或 `172.20.0.114:32724` 或者 `172.20.0.115:32724` 都可以得到nginx的页面。



图片 - *welcome nginx*

参考

[Kubelet 的认证授权]

安装kubedns插件

官方的yaml文件目录：`kubernetes/cluster/addons/dns`。

该插件直接使用kubernetes部署，官方的配置文件中包含以下镜像：

```
gcr.io/google_containers/k8s-dns-dnsmasq-nanny-amd64:1.14.1
gcr.io/google_containers/k8s-dns-kube-dns-amd64:1.14.1
gcr.io/google_containers/k8s-dns-sidecar-amd64:1.14.1
```

我clone了上述镜像，上传到我的私有镜像仓库：

```
harbor-001.jimmysong.io/library/k8s-dns-dnsmasq-nanny-amd64:1.14
.1
harbor-001.jimmysong.io/library/k8s-dns-kube-dns-amd64:1.14.1
harbor-001.jimmysong.io/library/k8s-dns-sidecar-amd64:1.14.1
```

同时上传了一份到时速云备份：

```
index.tenxcloud.com/jimmy/k8s-dns-dnsmasq-nanny-amd64:1.14.1
index.tenxcloud.com/jimmy/k8s-dns-kube-dns-amd64:1.14.1
index.tenxcloud.com/jimmy/k8s-dns-sidecar-amd64:1.14.1
```

以下yaml配置文件中使用的是私有镜像仓库中的镜像。

```
kubedns-cm.yaml
kubedns-sa.yaml
kubedns-controller.yaml
kubedns-svc.yaml
```

已经修改好的 yaml 文件见：../manifests/kubedns

系统预定义的 RoleBinding

预定义的 RoleBinding `system:kube-dns` 将 kube-system 命名空间的 `kube-dns` ServiceAccount 与 `system:kube-dns` Role 绑定， 该 Role 具有访问 kube-apiserver DNS 相关 API 的权限；

```
$ kubectl get clusterrolebindings system:kube-dns -o yaml
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: ClusterRoleBinding
metadata:
  annotations:
    rbac.authorization.kubernetes.io/autoupdate: "true"
  creationTimestamp: 2017-04-11T11:20:42Z
  labels:
    kubernetes.io/bootstrapping: rbac-defaults
  name: system:kube-dns
  resourceVersion: "58"
  selfLink: /apis/rbac.authorization.k8s.io/v1beta1/clusterroleb
indingssystem%3Akube-dns
  uid: e61f4d92-1ea8-11e7-8cd7-f4e9d49f8ed0
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: system:kube-dns
subjects:
- kind: ServiceAccount
  name: kube-dns
  namespace: kube-system
```

`kubedns-controller.yaml` 中定义的 Pods 时使用了 `kubedns-sa.yaml` 文件定义的 `kube-dns` ServiceAccount，所以具有访问 kube-apiserver DNS 相关 API 的权限。

配置 kube-dns ServiceAccount

无需修改。

配置 `kube-dns` 服务

```
$ diff kubedns-svc.yaml.base kubedns-svc.yaml
30c30
<   clusterIP: __PILLAR__DNS__SERVER__
---
>   clusterIP: 10.254.0.2
```

- spec.clusterIP = 10.254.0.2，即明确指定了 kube-dns Service IP，这个 IP 需要和 kubelet 的 `--cluster-dns` 参数值一致；

配置 `kube-dns` Deployment

```
$ diff kubedns-controller.yaml.base kubedns-controller.yaml
58c58
<         image: gcr.io/google_containers/k8s-dns-kube-dns-amd64
:1.14.1
---
>         image: harbor-001.jimmysong.io/library/k8s-dns-kube-dn
s-amd64:v1.14.1
88c88
<         - --domain=__PILLAR__DNS__DOMAIN__.
---
>         - --domain=cluster.local.
92c92
<         __PILLAR__FEDERATIONS__DOMAIN__MAP__
---
>         #__PILLAR__FEDERATIONS__DOMAIN__MAP__
110c110
<         image: gcr.io/google_containers/k8s-dns-dnsmasq-nanny-
amd64:1.14.1
---
>         image: harbor-001.jimmysong.io/library/k8s-dns-dnsmasq
-nanny-amd64:v1.14.1
129c129
<         - --server=/__PILLAR__DNS__DOMAIN__/127.0.0.1#10053
---
>         - --server=/cluster.local./127.0.0.1#10053
148c148
<         image: gcr.io/google_containers/k8s-dns-sidecar-amd64:
1.14.1
---
>         image: harbor-001.jimmysong.io/library/k8s-dns-sidecar
-amd64:v1.14.1
161,162c161,162
<         - --probe=kubedns,127.0.0.1:10053,kubernetes.default.s
vc.__PILLAR__DNS__DOMAIN__,5,A
<         - --probe=dnsmasq,127.0.0.1:53,kubernetes.default.svc.
__PILLAR__DNS__DOMAIN__,5,A
---
>         - --probe=kubedns,127.0.0.1:10053,kubernetes.default.s
vc.cluster.local.,5,A
>         - --probe=dnsmasq,127.0.0.1:53,kubernetes.default.svc.
cluster.local.,5,A
```

- 使用系统已经做了 RoleBinding 的 `kube-dns` ServiceAccount，该账户具有访问 kube-apiserver DNS 相关 API 的权限；

执行所有定义文件

```
$ pwd
/root/kubedns
$ ls *.yaml
kubedns-cm.yaml  kubedns-controller.yaml  kubedns-sa.yaml  kubed
ns-svc.yaml
$ kubectl create -f .
```

检查 kubedns 功能

新建一个 Deployment

```
$ cat  my-nginx.yaml
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: my-nginx
spec:
  replicas: 2
  template:
    metadata:
      labels:
        run: my-nginx
    spec:
      containers:
      - name: my-nginx
        image: harbor-001.jimmysong.io/library/nginx:1.9
        ports:
        - containerPort: 80
$ kubectl create -f my-nginx.yaml
```

Export 该 Deployment, 生成 `my-nginx` 服务

```
$ kubectl expose deploy my-nginx
$ kubectl get services --all-namespaces |grep my-nginx
default       my-nginx     10.254.179.239   <none>        80/TCP
          42m
```

创建另一个 Pod，查看 `/etc/resolv.conf` 是否包含 `kubelet` 配置的 `--cluster-dns` 和 `--cluster-domain`，是否能够将服务 `my-nginx` 解析到 Cluster IP `10.254.179.239`。

```
$ kubectl create -f nginx-pod.yaml
$ kubectl exec  nginx -i -t -- /bin/bash
root@nginx:/# cat /etc/resolv.conf
nameserver 10.254.0.2
search default.svc.cluster.local. svc.cluster.local. cluster.loc
al. jimmysong.io
options ndots:5

root@nginx:/# ping my-nginx
PING my-nginx.default.svc.cluster.local (10.254.179.239): 56 dat
a bytes
76 bytes from 119.147.223.109: Destination Net Unreachable
^C--- my-nginx.default.svc.cluster.local ping statistics ---

root@nginx:/# ping kubernetes
PING kubernetes.default.svc.cluster.local (10.254.0.1): 56 data
bytes
^C--- kubernetes.default.svc.cluster.local ping statistics ---
11 packets transmitted, 0 packets received, 100% packet loss

root@nginx:/# ping kube-dns.kube-system.svc.cluster.local
PING kube-dns.kube-system.svc.cluster.local (10.254.0.2): 56 dat
a bytes
^C--- kube-dns.kube-system.svc.cluster.local ping statistics ---
6 packets transmitted, 0 packets received, 100% packet loss
```

从结果来看，service名称可以正常解析。

注意：直接ping ClusterIP是ping不通的，ClusterIP是根据**IPtables**路由到服务的endpoint上，只有结合ClusterIP加端口才能访问到对应的服务。

安装dashboard插件

> 注意：本文档中安装的是kubernetes dashboard v1.6.0，安装新版的dashboard请参考升级dashboard。

官方文件目录：https://github.com/kubernetes/kubernetes/tree/master/cluster/addons/dashboard

我们使用的文件如下：

```
$ ls *.yaml
dashboard-controller.yaml  dashboard-service.yaml dashboard-rbac
.yaml
```

已经修改好的 yaml 文件见：../manifests/dashboard

文件中的kubernetes-dashboard-amd64镜像为本地镜像地址需要修改为对应的镜像地址和版本：

kubernetes 1.7.11 可以使用此镜像地址： registry.cn-qingdao.aliyuncs.com/haitao/kubernetes-dashboard-amd64:v1.7.0 替换 dashboard-controller.yaml 文件中的镜像地址

由于 `kube-apiserver` 启用了 `RBAC` 授权，而官方源码目录的 `dashboard-controller.yaml` 没有定义授权的 ServiceAccount，所以后续访问 API server 的 API 时会被拒绝，web中提示：

```
Forbidden (403)

User "system:serviceaccount:kube-system:default" cannot list job
s.batch in the namespace "default". (get jobs.batch)
```

增加了一个 `dashboard-rbac.yaml` 文件，定义一个名为 dashboard 的 ServiceAccount，然后将它和 Cluster Role view 绑定，如下：

```yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  name: dashboard
  namespace: kube-system
---
kind: ClusterRoleBinding
apiVersion: rbac.authorization.k8s.io/v1beta1
metadata:
  name: dashboard
subjects:
  - kind: ServiceAccount
    name: dashboard
    namespace: kube-system
roleRef:
  kind: ClusterRole
  name: cluster-admin
  apiGroup: rbac.authorization.k8s.io
```

然后使用 `kubectl apply -f dashboard-rbac.yaml` 创建。

配置dashboard-service

```bash
$ diff dashboard-service.yaml.orig dashboard-service.yaml
10a11
>   type: NodePort
```

- 指定端口类型为 NodePort，这样外界可以通过地址 nodeIP:nodePort 访问 dashboard；

配置dashboard-controller

```bash
$ diff dashboard-controller.yaml.orig dashboard-controller.yaml
23c23
<         image: gcr.io/google_containers/kubernetes-dashboard-a
md64:v1.6.0
---
>         image: harbor-001.jimmysong.io/library/kubernetes-dash
board-amd64:v1.6.0
```

执行所有定义文件

```
$ pwd
/root/kubernetes/cluster/addons/dashboard
$ ls *.yaml
dashboard-controller.yaml  dashboard-service.yaml
$ kubectl create -f  .
service "kubernetes-dashboard" created
deployment "kubernetes-dashboard" created
```

检查执行结果

查看分配的 NodePort

```
$ kubectl get services kubernetes-dashboard -n kube-system
NAME                   CLUSTER-IP       EXTERNAL-IP   PORT(S)
     AGE
kubernetes-dashboard   10.254.224.130   <nodes>       80:30312/T
CP    25s
```

- NodePort 30312映射到 dashboard pod 80端口；

检查 controller

```
$ kubectl get deployment kubernetes-dashboard  -n kube-system
NAME                   DESIRED   CURRENT   UP-TO-DATE   AVAILABL
E    AGE
kubernetes-dashboard   1         1         1            1
    3m
$ kubectl get pods  -n kube-system | grep dashboard
kubernetes-dashboard-1339745653-pmn6z   1/1       Running   0
       4m
```

访问dashboard

有以下三种方式：

- kubernetes-dashboard 服务暴露了 NodePort，可以使用 `http://NodeIP:nodePort` 地址访问 dashboard
- 通过 API server 访问 dashboard（https 6443端口和http 8080端口方式）
- 通过 kubectl proxy 访问 dashboard

通过 kubectl proxy 访问 dashboard

启动代理

```
$ kubectl proxy --address='172.20.0.113' --port=8086 --accept-ho
sts='^*$'
Starting to serve on 172.20.0.113:8086
```

- 需要指定 `--accept-hosts` 选项，否则浏览器访问 dashboard 页面时提示 “Unauthorized”；

浏览器访问 URL：http://172.20.0.113:8086/ui 自动跳转到：http://172.20.0.113:8086/api/v1/proxy/namespaces/kube-system/services/kubernetes-dashboard/#/workload?namespace=default

通过 API server 访问dashboard

获取集群服务地址列表

```
$ kubectl cluster-info
Kubernetes master is running at https://172.20.0.113:6443
KubeDNS is running at https://172.20.0.113:6443/api/v1/proxy/nam
espaces/kube-system/services/kube-dns
kubernetes-dashboard is running at https://172.20.0.113:6443/api
/v1/proxy/namespaces/kube-system/services/kubernetes-dashboard
```

浏览器访问 URL：https://172.20.0.113:6443/api/v1/proxy/namespaces/kube-system/services/kubernetes-dashboard（浏览器会提示证书验证，因为通过加密通道，以改方式访问的话，需要提前导入证书到你的计算机中）。这是我当时在这遇到的坑：通过 kube-apiserver 访问dashboard，提示User "system:anonymous" cannot proxy services in the namespace "kube-system". #5，已经解决。

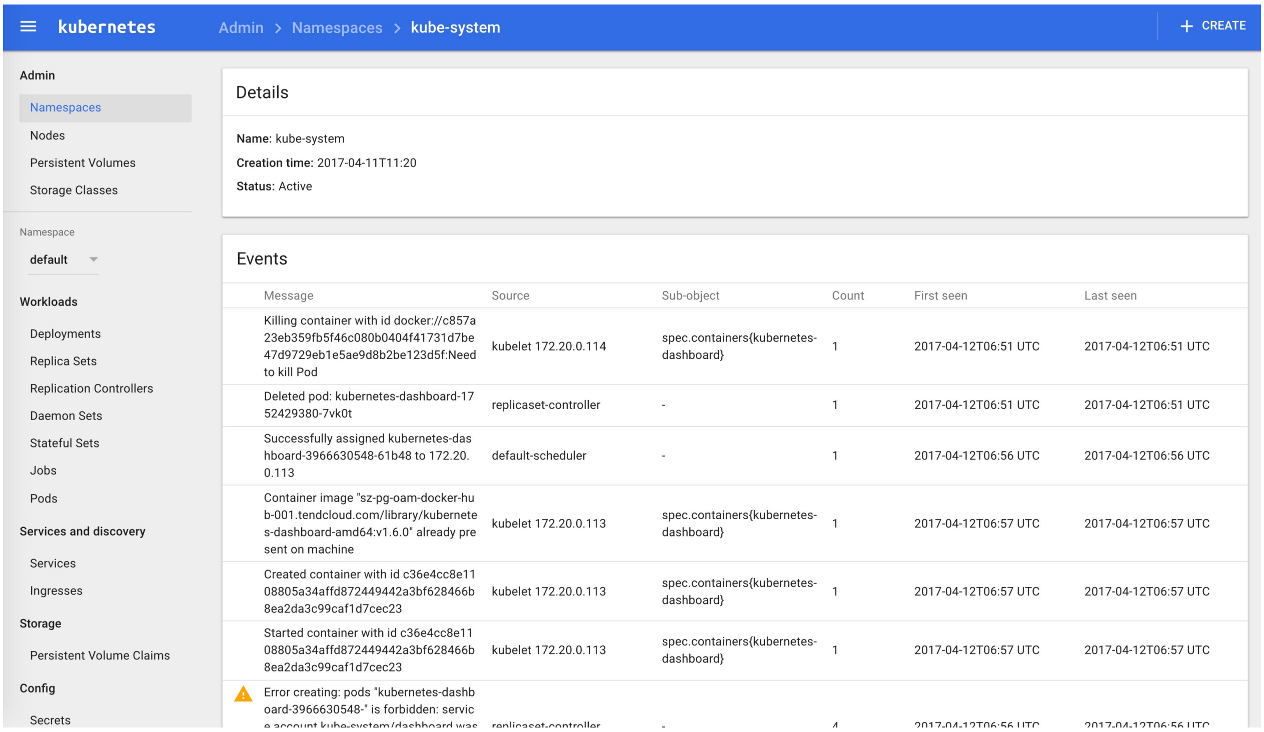
导入证书

将生成的admin.pem证书转换格式

```
openssl pkcs12 -export -in admin.pem  -out admin.p12 -inkey admi
n-key.pem
```

将生成的 `admin.p12` 证书导入的你的电脑，导出的时候记住你设置的密码，导入的时候还要用到。

如果你不想使用**https**的话，可以直接访问insecure port 8080端口：http://172.20.0.113:8080/api/v1/proxy/namespaces/kube-system/services/kubernetes-dashboard



图片 - *kubernetes dashboard*

由于缺少 Heapster 插件，当前 dashboard 不能展示 Pod、Nodes 的 CPU、内存等 metric 图形。

更新

Kubernetes 1.6 版本的 dashboard 的镜像已经到了 v1.6.3 版本，我们可以使用下面的方式更新。

修改 `dashboard-controller.yaml` 文件中的镜像的版本将 `v1.6.0` 更改为 `v1.6.3`。

```
image: harbor-001.jimmysong.io/library/kubernetes-dashboard-amd64
:v1.6.3
```

然后执行下面的命令：

```
kubectl apply -f dashboard-controller.yaml
```

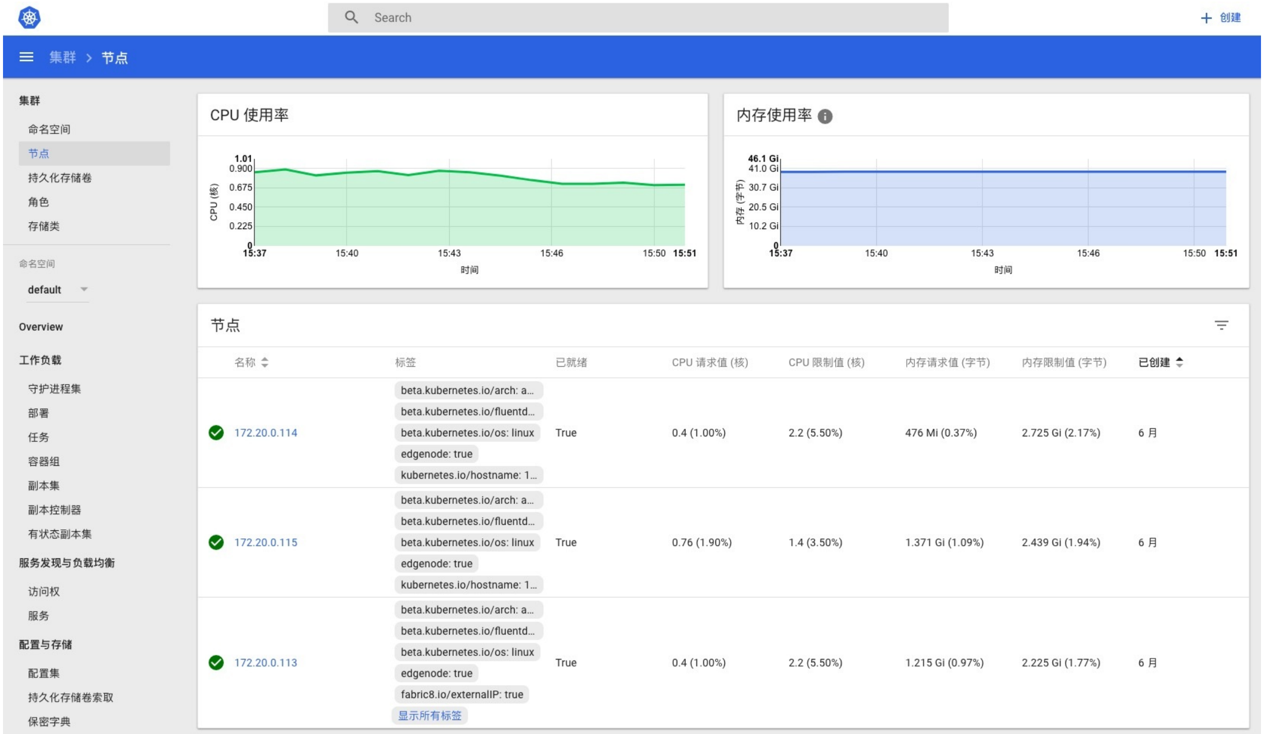
即可在线更新 dashboard 的版本。

监听 dashboard Pod 的状态可以看到：

```
kubernetes-dashboard-215087767-2jsgd   0/1       Pending   0
     0s
kubernetes-dashboard-3966630548-0jj1j   1/1       Terminating
0         1d
kubernetes-dashboard-215087767-2jsgd   0/1       Pending   0
     0s
kubernetes-dashboard-3966630548-0jj1j   1/1       Terminating
0         1d
kubernetes-dashboard-215087767-2jsgd   0/1       ContainerCreati
ng   0          0s
kubernetes-dashboard-3966630548-0jj1j   0/1       Terminating
0         1d
kubernetes-dashboard-3966630548-0jj1j   0/1       Terminating
0         1d
kubernetes-dashboard-215087767-2jsgd   1/1       Running   0
     6s
kubernetes-dashboard-3966630548-0jj1j   0/1       Terminating
0         1d
kubernetes-dashboard-3966630548-0jj1j   0/1       Terminating
0         1d
kubernetes-dashboard-3966630548-0jj1j   0/1       Terminating
0         1d
```

新的 Pod 的启动了，旧的 Pod 被终结了。

Dashboard 的访问地址不变，重新访问 http://172.20.0.113:8080/api/v1/proxy/namespaces/kube-system/services/kubernetes-dashboard，可以看到新版的界面：



图片 - *V1.6.3*版本的*dashboard*界面

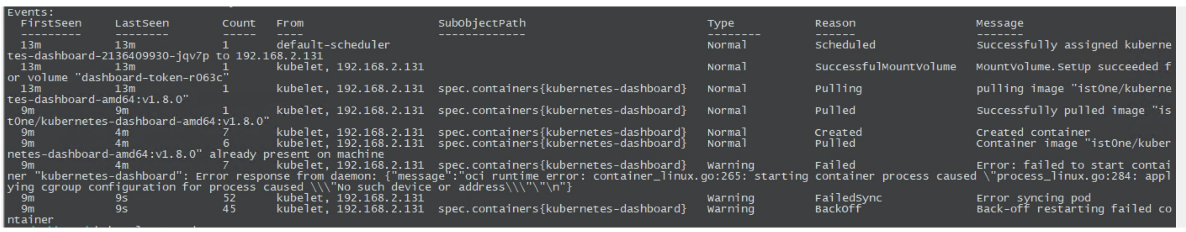
新版本中最大的变化是增加了进入容器内部的入口，可以在页面上进入到容器内部操作，同时又增加了一个搜索框。

关于如何将dashboard从1.6版本升级到1.7版本请参考升级dashboard。

问题

1. 按照教程安装后，发现dashboard pod 无法启动

```
kubectl -n kube-system describe pod dashboard-xxxxxxx
```



图片 - *pod*无法正常启动

可以尝试删除所有相关“资源”再重试一次，如：secret、serviceaccount、service、pod、deployment

参考

- WebUI(Dashboard) 文档

安装heapster插件

准备镜像

官方镜像保存在 gcr.io 中需要翻墙才能下载，为了方便大家使用，我下载后放到了时速云中，为公有镜像供大家下载。

- index.tenxcloud.com/jimmy/heapster-amd64:v1.3.0-beta.1
- index.tenxcloud.com/jimmy/heapster-influxdb-amd64:v1.1.1
- index.tenxcloud.com/jimmy/heapster-grafana-amd64:v4.0.2

准备YAML文件

到 heapster release 页面 下载最新版本的 heapster。

```
wget https://github.com/kubernetes/heapster/archive/v1.3.0.zip
unzip v1.3.0.zip
mv v1.3.0.zip heapster-1.3.0
```

文件目录： `heapster-1.3.0/deploy/kube-config/influxdb`

```
$ cd heapster-1.3.0/deploy/kube-config/influxdb
$ ls *.yaml
grafana-deployment.yaml  grafana-service.yaml  heapster-deployme
nt.yaml  heapster-service.yaml  influxdb-deployment.yaml  influx
db-service.yaml heapster-rbac.yaml
```

我们自己创建了heapster的rbac配置 `heapster-rbac.yaml` 。

已经修改好的 yaml 文件见：../manifests/heapster

配置 grafana-deployment

```
$ diff grafana-deployment.yaml.orig grafana-deployment.yaml
16c16
<           image: gcr.io/google_containers/heapster-grafana-amd64
:v4.0.2
---
>           image: harbor-001.jimmysong.io/library/heapster-grafan
a-amd64:v4.0.2
40,41c40,41
<             # value: /api/v1/proxy/namespaces/kube-system/servic
es/monitoring-grafana/
<             value: /
---
>             value: /api/v1/proxy/namespaces/kube-system/services
/monitoring-grafana/
>             #value: /
```

- 如果后续使用 kube-apiserver 或者 kubectl proxy 访问 grafana dashboard，则必须将 `GF_SERVER_ROOT_URL` 设置为 `/api/v1/proxy/namespaces/kube-system/services/monitoring-grafana/`，否则后续访问grafana时访问时提示找不到 `http://172.20.0.113:8086/api/v1/proxy/namespaces/kube-system/services/monitoring-grafana/api/dashboards/home` 页面；

配置 heapster-deployment

```
$ diff heapster-deployment.yaml.orig heapster-deployment.yaml
16c16
<           image: gcr.io/google_containers/heapster-amd64:v1.3.0-
beta.1
---
>           image: harbor-001.jimmysong.io/library/heapster-amd64:
v1.3.0-beta.1
```

配置 influxdb-deployment

influxdb 官方建议使用命令行或 HTTP API 接口来查询数据库，从 v1.1.0 版本开始默认关闭 admin UI，将在后续版本中移除 admin UI 插件。

开启镜像中 admin UI的办法如下：先导出镜像中的 influxdb 配置文件，开启 admin 插件后，再将配置文件内容写入 ConfigMap，最后挂载到镜像中，达到覆盖原始配置的目的：

注意：manifests 目录已经提供了 修改后的 ConfigMap 定义文件

```
$ # 导出镜像中的 influxdb 配置文件
$ docker run --rm --entrypoint 'cat'  -ti lvanneo/heapster-influ
xdb-amd64:v1.1.1 /etc/config.toml >config.toml.orig
$ cp config.toml.orig config.toml
$ # 修改：启用 admin 接口
$ vim config.toml
$ diff config.toml.orig config.toml
35c35
<   enabled = false
---
>   enabled = true
$ # 将修改后的配置写入到 ConfigMap 对象中
$ kubectl create configmap influxdb-config --from-file=config.to
ml  -n kube-system
configmap "influxdb-config" created
$ # 将 ConfigMap 中的配置文件挂载到 Pod 中，达到覆盖原始配置的目的
$ diff influxdb-deployment.yaml.orig influxdb-deployment.yaml
16c16
<         image: gcr.io/google_containers/heapster-influxdb-amd6
4:v1.1.1
---
>         image: harbor-001.jimmysong.io/library/heapster-influx
db-amd64:v1.1.1
19a20,21
>         - mountPath: /etc/
>           name: influxdb-config
22a25,27
>       - name: influxdb-config
>         configMap:
>           name: influxdb-config
```

配置 monitoring-influxdb Service

```
$ diff influxdb-service.yaml.orig influxdb-service.yaml
12a13
>   type: NodePort
15a17,20
>     name: http
>   - port: 8083
>     targetPort: 8083
>     name: admin
```

- 定义端口类型为 NodePort，额外增加了 admin 端口映射，用于后续浏览器访问 influxdb 的 admin UI 界面；

执行所有定义文件

```
$ pwd
/root/heapster-1.3.0/deploy/kube-config/influxdb
$ ls *.yaml
grafana-service.yaml      heapster-rbac.yaml     influxdb-cm.yam
l           influxdb-service.yaml
grafana-deployment.yaml  heapster-deployment.yaml  heapster-serv
ice.yaml  influxdb-deployment.yaml
$ kubectl create -f  .
deployment "monitoring-grafana" created
service "monitoring-grafana" created
deployment "heapster" created
serviceaccount "heapster" created
clusterrolebinding "heapster" created
service "heapster" created
configmap "influxdb-config" created
deployment "monitoring-influxdb" created
service "monitoring-influxdb" created
```

检查执行结果

检查 Deployment

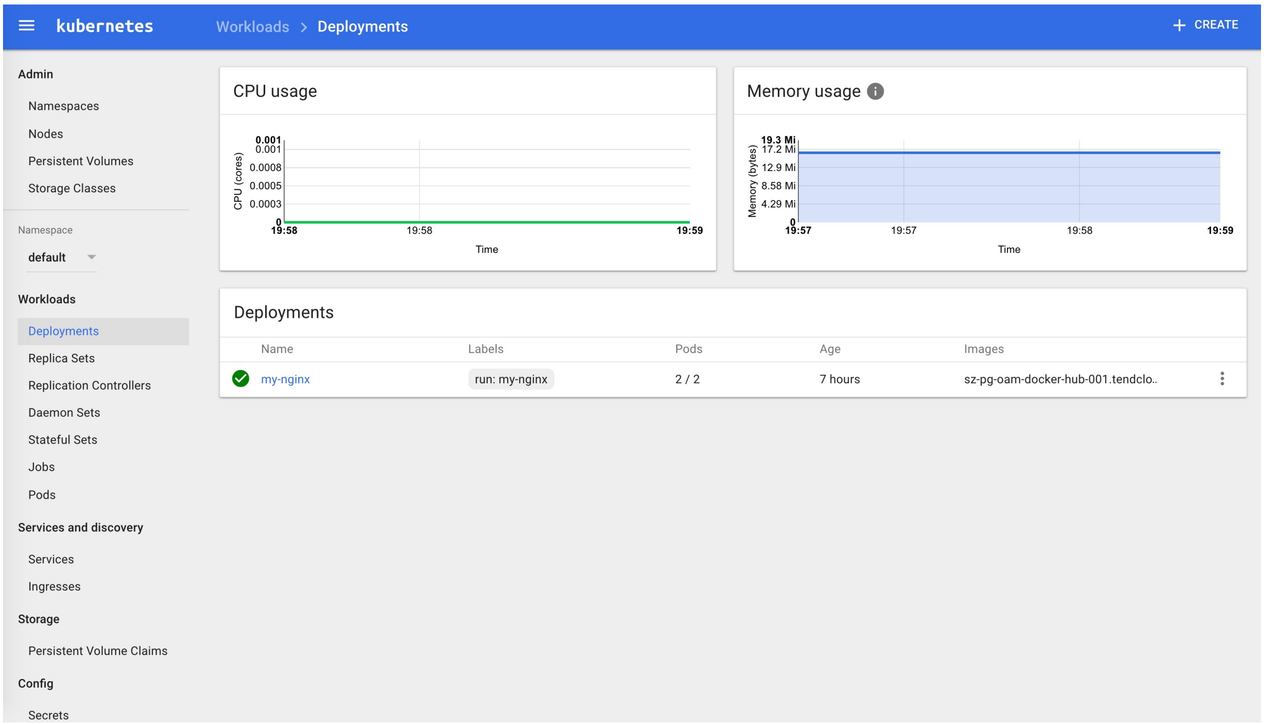
```
$ kubectl get deployments -n kube-system | grep -E 'heapster|mon
itoring'
heapster               1         1         1            1
    2m
monitoring-grafana     1         1         1            1
    2m
monitoring-influxdb    1         1         1            1
    2m
```

检查 Pods

```
$ kubectl get pods -n kube-system | grep -E 'heapster|monitoring'

heapster-110704576-gpg8v                1/1       Running   0
       2m
monitoring-grafana-2861879979-9z89f     1/1       Running   0
       2m
monitoring-influxdb-1411048194-lzrpc    1/1       Running   0
       2m
```

检查 kubernets dashboard 界面，看是显示各 Nodes、Pods 的 CPU、内存、负载等利用率曲线图；



图片 - *dashboard-heapster*

访问 grafana

1. 通过 kube-apiserver 访问：

   获取 monitoring-grafana 服务 URL

```
$ kubectl cluster-info
Kubernetes master is running at https://172.20.0.113:6443
Heapster is running at https://172.20.0.113:6443/api/v1/proxy/namespaces/kube-system/services/heapster
KubeDNS is running at https://172.20.0.113:6443/api/v1/proxy/namespaces/kube-system/services/kube-dns
kubernetes-dashboard is running at https://172.20.0.113:6443/api/v1/proxy/namespaces/kube-system/services/kubernetes-dashboard
monitoring-grafana is running at https://172.20.0.113:6443/api/v1/proxy/namespaces/kube-system/services/monitoring-grafana
monitoring-influxdb is running at https://172.20.0.113:6443/api/v1/proxy/namespaces/kube-system/services/monitoring-influxdb

To further debug and diagnose cluster problems, use 'kubectl cluster-info dump'.
```

浏览器访问 URL： `http://172.20.0.113:8080/api/v1/proxy/namespaces/kube-system/services/monitoring-grafana`

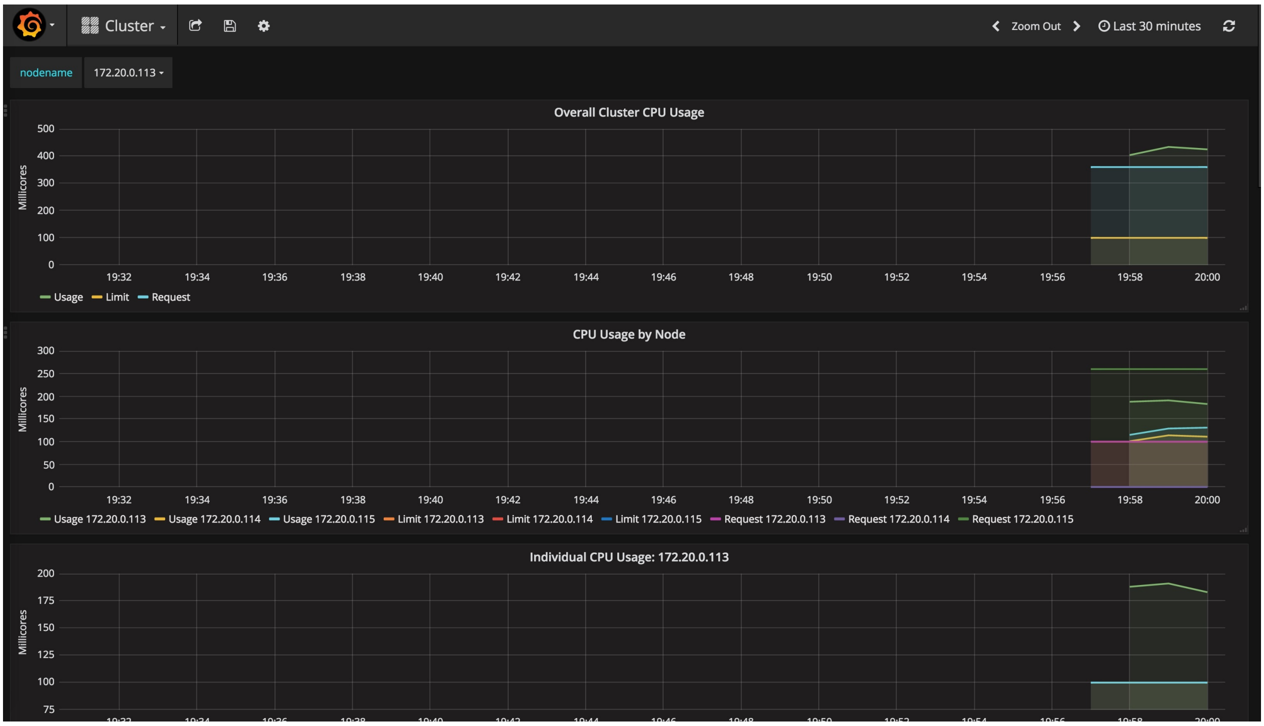
2. 通过 kubectl proxy 访问：

   创建代理

```
$ kubectl proxy --address='172.20.0.113' --port=8086 --accept-hosts='^*$'
Starting to serve on 172.20.0.113:8086
```

浏览器访问

URL： `http://172.20.0.113:8086/api/v1/proxy/namespaces/kube-system/services/monitoring-grafana`



图片 - *grafana*

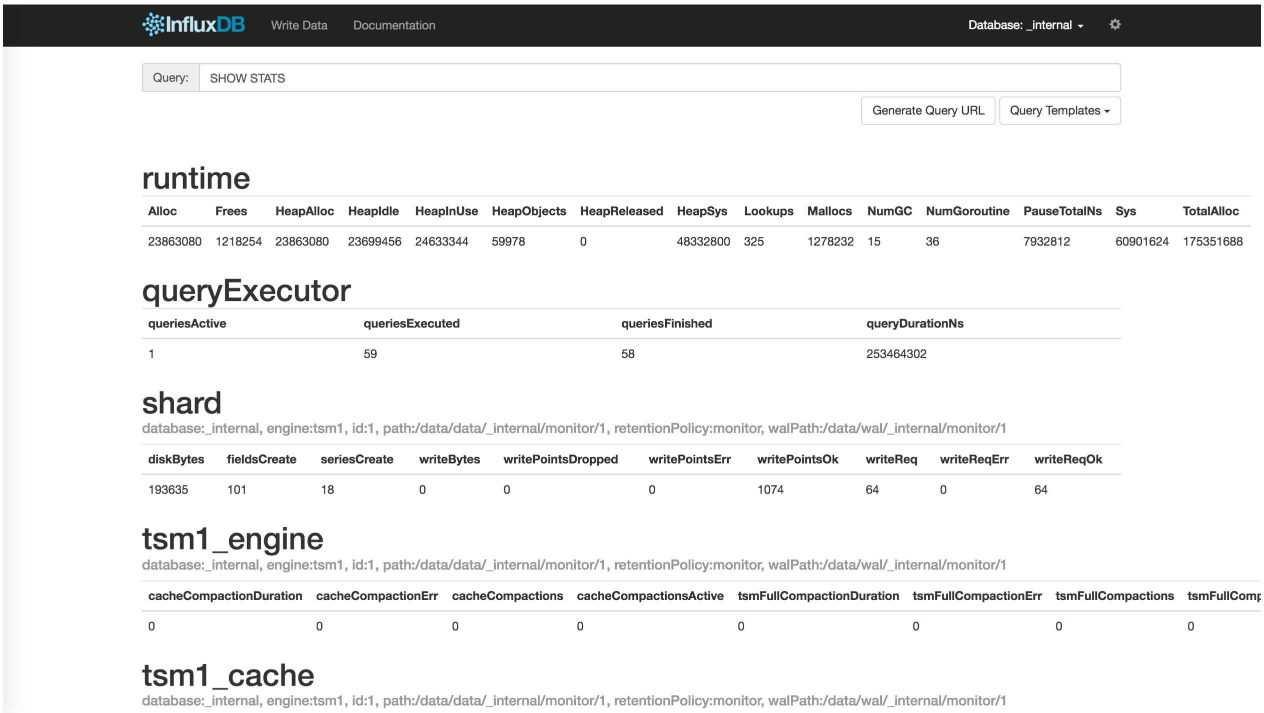
访问 influxdb admin UI

获取 influxdb http 8086 映射的 NodePort

```
$ kubectl get svc -n kube-system|grep influxdb
monitoring-influxdb    10.254.22.46    <nodes>       8086:32299/
TCP,8083:30269/TCP   9m
```

通过 kube-apiserver 的非安全端口访问 influxdb 的 admin UI 界面： `http://172.20.0.113:8080/api/v1/proxy/namespaces/kube-system/services/monitoring-influxdb:8083/`

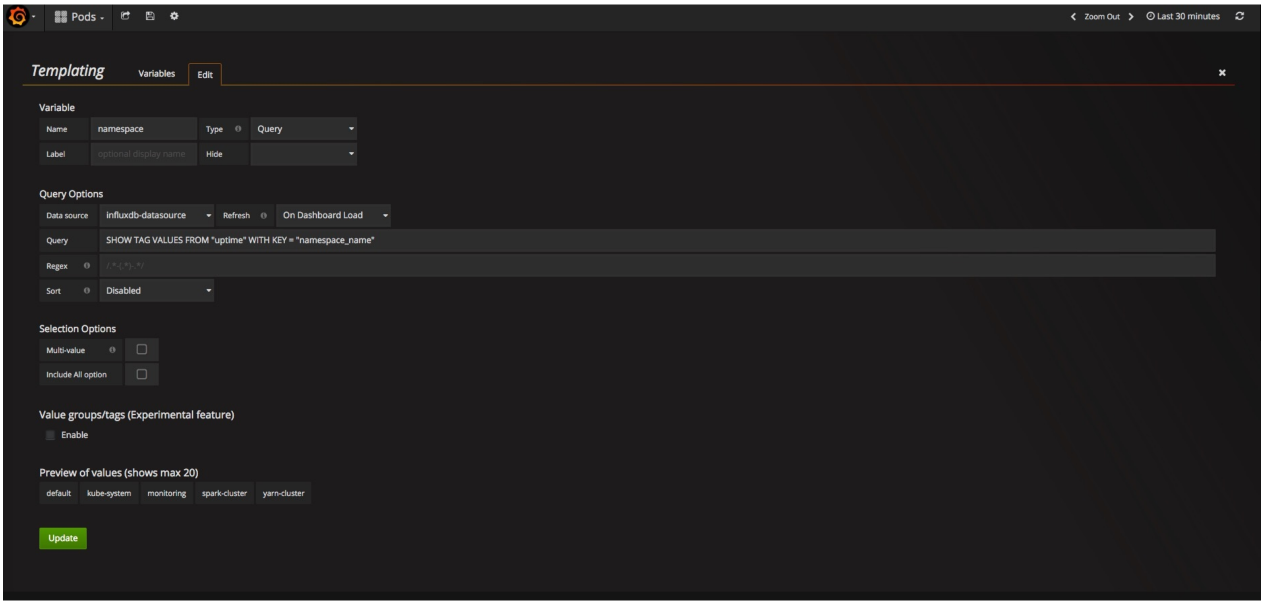
在页面的 “Connection Settings” 的 Host 中输入 node IP， Port 中输入 8086 映射的 nodePort 如上面的 32299，点击 “Save” 即可（我的集群中的地址是 172.20.0.113:32299）：



图片 - *kubernetes-influxdb-heapster*

注意

在安装好 Grafana 之后我们使用的是默认的 template 配置，页面上的 namespace 选择里只有 `default` 和 `kube-system`，并不是说其他的 namespace 里的指标没有得到监控，只是我们没有在 Grafana 中开启他它们的显示而已。见 Cannot see other namespaces except, kube-system and default #1279。



图片 - 修改grafana模板

将 Templating 中的 namespace 的 Data source 设置为 influxdb-datasource，Refresh 设置为 on Dashboard Load 保存设置，刷新浏览器，即可看到其他 namespace 选项。

参考

[使用Heapster获取集群对象的metric数据]

安装EFK插件

我们通过在每台node上部署一个以DaemonSet方式运行的fluentd来收集每台node上的日志。Fluentd将docker日志目录 `/var/lib/docker/containers` 和 `/var/log` 目录挂载到Pod中，然后Pod会在node节点的 `/var/log/pods` 目录中创建新的目录，可以区别不同的容器日志输出，该目录下有一个日志文件链接到 `/var/lib/docker/contianers` 目录下的容器日志输出。

官方文件目录： `cluster/addons/fluentd-elasticsearch`

```
$ ls *.yaml
es-controller.yaml  es-service.yaml  fluentd-es-ds.yaml  kibana-controller.yaml  kibana-service.yaml efk-rbac.yaml
```

同样EFK服务也需要一个 `efk-rbac.yaml` 文件，配置serviceaccount为 `efk` 。

已经修改好的 yaml 文件见：../manifests/EFK

配置 es-controller.yaml

```
$ diff es-controller.yaml.orig es-controller.yaml
24c24
<       - image: gcr.io/google_containers/elasticsearch:v2.4.1-2
---
>       - image: harbor-001.jimmysong.io/library/elasticsearch:v2.4.1-2
```

配置 es-service.yaml

无需配置；

配置 fluentd-es-ds.yaml

```
$ diff fluentd-es-ds.yaml.orig fluentd-es-ds.yaml
26c26
<         image: gcr.io/google_containers/fluentd-elasticsearch:
1.22
---
>         image: harbor-001.jimmysong.io/library/fluentd-elastic
search:1.22
```

配置 kibana-controller.yaml

```
$ diff kibana-controller.yaml.orig kibana-controller.yaml
22c22
<         image: gcr.io/google_containers/kibana:v4.6.1-1
---
>         image: harbor-001.jimmysong.io/library/kibana:v4.6.1-1
```

给 Node 设置标签

定义 DaemonSet `fluentd-es-v1.22` 时设置了 nodeSelector `beta.kubernetes.io/fluentd-ds-ready=true` ，所以需要在期望运行 fluentd 的 Node 上设置该标签；

```
$ kubectl get nodes
NAME          STATUS    AGE       VERSION
172.20.0.113   Ready     1d        v1.6.0

$ kubectl label nodes 172.20.0.113 beta.kubernetes.io/fluentd-ds
-ready=true
node "172.20.0.113" labeled
```

给其他两台node打上同样的标签。

执行定义文件

```
$ kubectl create -f .
serviceaccount "efk" created
clusterrolebinding "efk" created
replicationcontroller "elasticsearch-logging-v1" created
service "elasticsearch-logging" created
daemonset "fluentd-es-v1.22" created
deployment "kibana-logging" created
service "kibana-logging" created
```

检查执行结果

```
$ kubectl get deployment -n kube-system|grep kibana
kibana-logging         1         1         1            1
    2m

$ kubectl get pods -n kube-system|grep -E 'elasticsearch|fluentd
|kibana'
elasticsearch-logging-v1-mlstp          1/1       Running   0
       1m
elasticsearch-logging-v1-nfbbf          1/1       Running   0
       1m
fluentd-es-v1.22-31sm0                  1/1       Running   0
       1m
fluentd-es-v1.22-bpgqs                  1/1       Running   0
       1m
fluentd-es-v1.22-qmn7h                  1/1       Running   0
       1m
kibana-logging-1432287342-0gdng         1/1       Running   0
       1m

$ kubectl get service  -n kube-system|grep -E 'elasticsearch|kib
ana'
elasticsearch-logging   10.254.77.62    <none>        9200/TCP
                      2m
kibana-logging          10.254.8.113    <none>        5601/TCP
                      2m
```

kibana Pod 第一次启动时会用较长时间(10-20分钟)来优化和 Cache 状态页面，可以 tailf 该 Pod 的日志观察进度：

```
$ kubectl logs kibana-logging-1432287342-0gdng -n kube-system -f
ELASTICSEARCH_URL=http://elasticsearch-logging:9200
server.basePath: /api/v1/proxy/namespaces/kube-system/services/k
ibana-logging
{"type":"log","@timestamp":"2017-04-12T13:08:06Z","tags":["info",
"optimize"],"pid":7,"message":"Optimizing and caching bundles fo
```

```
r kibana and statusPage. This may take a few minutes"}
{"type":"log","@timestamp":"2017-04-12T13:18:17Z","tags":["info",
"optimize"],"pid":7,"message":"Optimization of bundles for kiban
a and statusPage complete in 610.40 seconds"}
{"type":"log","@timestamp":"2017-04-12T13:18:17Z","tags":["statu
s","plugin:kibana@1.0.0","info"],"pid":7,"state":"green","messag
e":"Status changed from uninitialized to green - Ready","prevSta
te":"uninitialized","prevMsg":"uninitialized"}
{"type":"log","@timestamp":"2017-04-12T13:18:18Z","tags":["statu
s","plugin:elasticsearch@1.0.0","info"],"pid":7,"state":"yellow",
"message":"Status changed from uninitialized to yellow - Waiting
 for Elasticsearch","prevState":"uninitialized","prevMsg":"unini
tialized"}
{"type":"log","@timestamp":"2017-04-12T13:18:19Z","tags":["statu
s","plugin:kbn_vislib_vis_types@1.0.0","info"],"pid":7,"state":"
green","message":"Status changed from uninitialized to green - R
eady","prevState":"uninitialized","prevMsg":"uninitialized"}
{"type":"log","@timestamp":"2017-04-12T13:18:19Z","tags":["statu
s","plugin:markdown_vis@1.0.0","info"],"pid":7,"state":"green","
message":"Status changed from uninitialized to green - Ready","p
revState":"uninitialized","prevMsg":"uninitialized"}
{"type":"log","@timestamp":"2017-04-12T13:18:19Z","tags":["statu
s","plugin:metric_vis@1.0.0","info"],"pid":7,"state":"green","me
ssage":"Status changed from uninitialized to green - Ready","pre
vState":"uninitialized","prevMsg":"uninitialized"}
{"type":"log","@timestamp":"2017-04-12T13:18:19Z","tags":["statu
s","plugin:spyModes@1.0.0","info"],"pid":7,"state":"green","mess
age":"Status changed from uninitialized to green - Ready","prevS
tate":"uninitialized","prevMsg":"uninitialized"}
{"type":"log","@timestamp":"2017-04-12T13:18:19Z","tags":["statu
s","plugin:statusPage@1.0.0","info"],"pid":7,"state":"green","me
ssage":"Status changed from uninitialized to green - Ready","pre
vState":"uninitialized","prevMsg":"uninitialized"}
{"type":"log","@timestamp":"2017-04-12T13:18:19Z","tags":["statu
s","plugin:table_vis@1.0.0","info"],"pid":7,"state":"green","mes
sage":"Status changed from uninitialized to green - Ready","prev
State":"uninitialized","prevMsg":"uninitialized"}
{"type":"log","@timestamp":"2017-04-12T13:18:19Z","tags":["liste
ning","info"],"pid":7,"message":"Server running at http://0.0.0.
0:5601"}
{"type":"log","@timestamp":"2017-04-12T13:18:24Z","tags":["statu
s","plugin:elasticsearch@1.0.0","info"],"pid":7,"state":"yellow",
"message":"Status changed from yellow to yellow - No existing Ki
bana index found","prevState":"yellow","prevMsg":"Waiting for El
asticsearch"}
{"type":"log","@timestamp":"2017-04-12T13:18:29Z","tags":["statu
s","plugin:elasticsearch@1.0.0","info"],"pid":7,"state":"green",
"message":"Status changed from yellow to green - Kibana index re
ady","prevState":"yellow","prevMsg":"No existing Kibana index fo
und"}
```

访问 kibana

1. 通过 kube-apiserver 访问：

   获取 monitoring-grafana 服务 URL

   ```
    $ kubectl cluster-info
    Kubernetes master is running at https://172.20.0.113:6443
    Elasticsearch is running at https://172.20.0.113:6443/api/v1/proxy/namespaces/kube-system/services/elasticsearch-logging
    Heapster is running at https://172.20.0.113:6443/api/v1/proxy/namespaces/kube-system/services/heapster
    Kibana is running at https://172.20.0.113:6443/api/v1/proxy/namespaces/kube-system/services/kibana-logging
    KubeDNS is running at https://172.20.0.113:6443/api/v1/proxy/namespaces/kube-system/services/kube-dns
    kubernetes-dashboard is running at https://172.20.0.113:6443/api/v1/proxy/namespaces/kube-system/services/kubernetes-dashboard
    monitoring-grafana is running at https://172.20.0.113:6443/api/v1/proxy/namespaces/kube-system/services/monitoring-grafana
    monitoring-influxdb is running at https://172.20.0.113:6443/api/v1/proxy/namespaces/kube-system/services/monitoring-influxdb
   ```

   浏览器访问 URL： `https://172.20.0.113:6443/api/v1/proxy/namespaces/kube-system/services/kibana-logging/app/kibana`

2. 通过 kubectl proxy 访问：

   创建代理

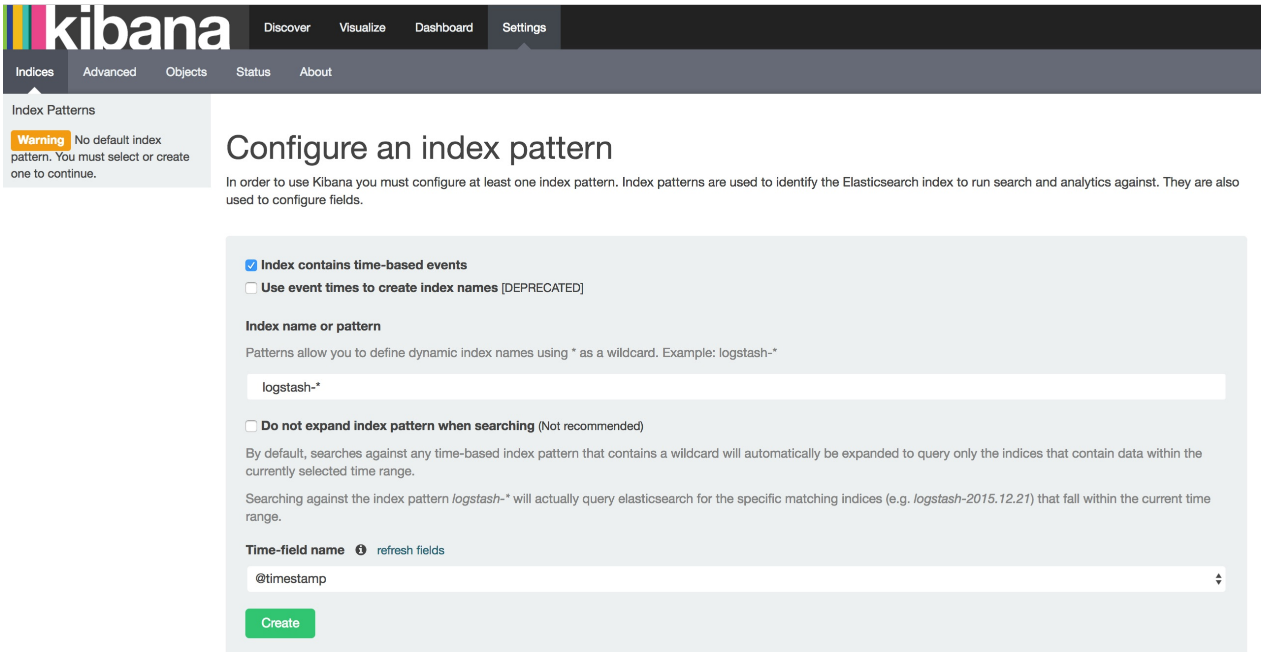
   ```
    $ kubectl proxy --address='172.20.0.113' --port=8086 --accept-hosts='^*$'
    Starting to serve on 172.20.0.113:8086
   ```

   浏览器访问 URL： `http://172.20.0.113:8086/api/v1/proxy/namespaces/kube-system/services/kibana-logging`

在 Settings -> Indices 页面创建一个 index（相当于 mysql 中的一个 database），选中 `Index contains time-based events`，使用默认的 `logstash-*` pattern，点击 `Create`；

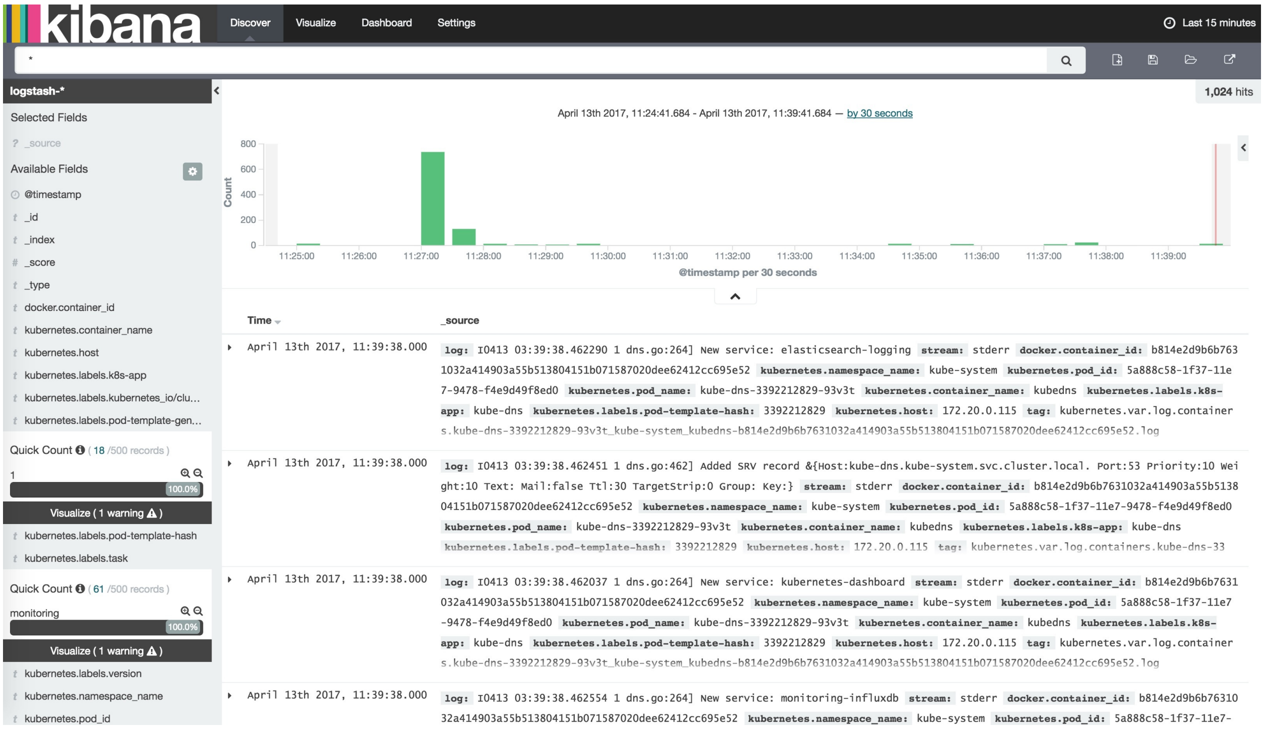
可能遇到的问题

如果你在这里发现Create按钮是灰色的无法点击，且Time-filed name中没有选项，fluentd要读取 `/var/log/containers/` 目录下的log日志，这些日志是从 `/var/lib/docker/containers/${CONTAINER_ID}/${CONTAINER_ID}-json.log` 链接过来的，查看你的docker配置，`—log-dirver` 需要设置为**json-file**格式，默认的可能是**journald**，参考[docker logging](docker logging)。



图片 - *es-setting*

创建Index后，可以在 `Discover` 下看到 ElasticSearch logging 中汇聚的日志；


kibana
Discover
Visualize
Dashboard
Settings
Last 15 minutes
logstash-*
1,024 hits
Selected Fields
_source
Available Fields
@timestamp
_id
_index
_score
_type
docker.container_id
kubernetes.container_name
kubernetes.host
kubernetes.labels.k8s-app
kubernetes.labels.kubernetes_io/clu...
kubernetes.labels.pod-template-gen...
Quick Count (18 /500 records)
Visualize (1 warning)
kubernetes.labels.pod-template-hash
kubernetes.labels.task
Quick Count (61 /500 records)
monitoring
Visualize (1 warning)
kubernetes.labels.version
kubernetes.namespace_name
kubernetes.pod_id
April 13th 2017, 11:24:41.684 - April 13th 2017, 11:39:41.684 — by 30 seconds
Count
@timestamp per 30 seconds
Time
_source
April 13th 2017, 11:39:38.000
April 13th 2017, 11:39:38.000
April 13th 2017, 11:39:38.000
April 13th 2017, 11:39:38.000

图片 - *es-home*

kubeadm

基本介绍

kubeadm 是一个工具包，可帮助您以简单，合理安全和可扩展的方式引导最佳实践Kubernetes群集。它还支持为您管理Bootstrap Tokens并升级/降级群集。

kubeadm的目标是建立一个通过Kubernetes一致性测试Kubernetes Conformance tests的最小可行集群 ，但不会安装其他功能插件。

它在设计上并未为您安装网络解决方案，需要用户自行安装第三方符合CNI的网络解决方案（如flanal，calico，canal等）。

kubeadm可以在多种设备上运行，可以是Linux笔记本电脑，虚拟机，物理/云服务器或Raspberry Pi。这使得kubeadm非常适合与不同种类的配置系统（例如Terraform，Ansible等）集成。

kubeadm是一种简单的方式让新用户开始尝试Kubernetes，也可能是第一次让现有用户轻松测试他们的应用程序并缝合到一起的方式，也可以作为其他生态系统中的构建块，或者具有更大范围的安装工具。

可以在支持安装deb或rpm软件包的操作系统上非常轻松地安装kubeadm。SIG集群生命周期SIG Cluster Lifecycle kubeadm的SIG相关维护者提供了预编译的这些软件包，也可以在其他操作系统上使用。

kubeadm 成熟度

分类	成熟度 **Level**
Command line UX	beta
Implementation	beta
Config file API	alpha
Self-hosting	alpha
kubeadm alpha subcommands	alpha
CoreDNS	alpha
DynamicKubeletConfig	alpha

kubeadm的整体功能状态为 **Beta**，即将在2018 年推向 **General Availability**（**GA**）。一些子功能（如自托管或配置文件API）仍在积极开发中。随着这个工具的发展，创建集群的实现可能会稍微改变，但总体实现应该相当稳定。根据 `kubeadm alpha` 定义，任何命令 都在alpha级别上受支持。

支持时间表

Kubernetes版本通常支持九个月，在此期间，如果发现严重的错误或安全问题，可能会从发布分支发布补丁程序版本。这里是最新的Kubernetes版本和支持时间表; 这也适用于 `kubeadm` .

Kubernetes version	Release month	End-of-life-month
v1.6.x	March 2017	December 2017
v1.7.x	June 2017	March 2018
v1.8.x	September 2017	June 2018
v1.9.x	December 2017	September 2018
v1.10.x	March 2018	December 2018

用kubeadm在Ubuntu上快速构建Kubernetes测试集群

本文将介绍如何在Ubuntu server 16.04版本上安装kubeadm，并利用kubeadm快速的在Ubuntu server 版本 16.04上构建一个kubernetes的基础的测试集群，用来做学习和测试用途，当前（2018-04-14）最新的版本是1.10.1。参考文档包括kubernetes官方网站的kubeadm安装文档以及利用kubeadm创建集群这两个文档。

生产用途的环境，需要考虑各个组件的高可用，建议参考Kubernetes的官方的相关的安装文档。

概述

本次安装建议至少4台服务器或者虚拟机，每台服务器4G内存，2个CPU核心以上，基本架构为1台master节点，3台slave节点。整个安装过程将在Ubuntu服务器上安装完kubeadm，以及安装kubernetes的基本集群，包括canal网络，另后台存储可参考本书的最佳实践中的存储管理内容。 本次安装一共4个节点，节点信息如下:

角色	主机名	IP地址
Master	Ubuntu-master	192.168.5.200
Slave	ubuntu-1	192.168.5.201
Slave	ubuntu-2	192.168.5.202
Slave	ubuntu-3	192.168.5.203

准备工作

- 默认方式安装Ubuntu Server 版本 16.04
- 配置主机名映射，每个节点

```
# cat /etc/hosts
127.0.0.1	localhost
192.168.0.200	Ubuntu-master
192.168.0.201	Ubuntu-1
192.168.0.202	Ubuntu-2
192.168.0.203	Ubuntu-3
```

- 如果连接gcr网站不方便，无法下载镜像，会导致安装过程卡住，可以下载我导出的镜像包，我导出的镜像网盘链接，解压缩以后是多个个tar包，使用 `docker load< xxxx.tar` 导入各个文件即可）。

在所有节点上安装kubeadm

查看apt安装源如下配置，使用阿里云的系统和kubernetes的源。

```
$ cat /etc/apt/sources.list
# 系统安装源
deb http://mirrors.aliyun.com/ubuntu/ xenial main restricted
deb http://mirrors.aliyun.com/ubuntu/ xenial-updates main restri
cted
deb http://mirrors.aliyun.com/ubuntu/ xenial universe
deb http://mirrors.aliyun.com/ubuntu/ xenial-updates universe
deb http://mirrors.aliyun.com/ubuntu/ xenial multiverse
deb http://mirrors.aliyun.com/ubuntu/ xenial-updates multiverse
deb http://mirrors.aliyun.com/ubuntu/ xenial-backports main rest
ricted universe multiverse
# kubeadm及kubernetes组件安装源
deb https://mirrors.aliyun.com/kubernetes/apt kubernetes-xenial
main
```

安装docker，可以使用系统源的的docker.io软件包，版本1.13.1，我的系统里是已经安装好最新的版本了。

```
# apt-get install docker.io
Reading package lists... Done
Building dependency tree
Reading state information... Done
docker.io is already the newest version (1.13.1-0ubuntu1~16.04.2)
.
0 upgraded, 0 newly installed, 0 to remove and 4 not upgraded.
```

更新源，可以不理会gpg的报错信息。

```
# apt-get update
Hit:1 http://mirrors.aliyun.com/ubuntu xenial InRelease
Hit:2 http://mirrors.aliyun.com/ubuntu xenial-updates InRelease
Hit:3 http://mirrors.aliyun.com/ubuntu xenial-backports InReleas
e
Get:4 https://mirrors.aliyun.com/kubernetes/apt kubernetes-xenia
l InRelease [8,993 B]
Ign:4 https://mirrors.aliyun.com/kubernetes/apt kubernetes-xenia
l InRelease
Fetched 8,993 B in 0s (20.7 kB/s)
Reading package lists... Done
W: GPG error: https://mirrors.aliyun.com/kubernetes/apt kubernet
es-xenial InRelease: The following signatures couldn't be verifi
ed because the public key is not available: NO_PUBKEY 6A030B21BA
07F4FB
W: The repository 'https://mirrors.aliyun.com/kubernetes/apt kub
ernetes-xenial InRelease' is not signed.
N: Data from such a repository can't be authenticated and is the
refore potentially dangerous to use.
N: See apt-secure(8) manpage for repository creation and user co
nfiguration details.
```

强制安装kubeadm，kubectl，kubelet软件包。

```
# apt-get install -y kubelet kubeadm kubectl --allow-unauthentic
ated
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following additional packages will be installed:
  kubernetes-cni socat
The following NEW packages will be installed:
  kubeadm kubectl kubelet kubernetes-cni socat
0 upgraded, 5 newly installed, 0 to remove and 4 not upgraded.
Need to get 56.9 MB of archives.
After this operation, 410 MB of additional disk space will be us
ed.
WARNING: The following packages cannot be authenticated!
  kubernetes-cni kubelet kubectl kubeadm
Authentication warning overridden.
Get:1 http://mirrors.aliyun.com/ubuntu xenial/universe amd64 soc
at amd64 1.7.3.1-1 [321 kB]
Get:2 https://mirrors.aliyun.com/kubernetes/apt kubernetes-xenia
l/main amd64 kubernetes-cni amd64 0.6.0-00 [5,910 kB]
Get:3 https://mirrors.aliyun.com/kubernetes/apt kubernetes-xenia
l/main amd64 kubelet amd64 1.10.1-00 [21.1 MB]
Get:4 https://mirrors.aliyun.com/kubernetes/apt kubernetes-xenia
l/main amd64 kubectl amd64 1.10.1-00 [8,906 kB]
Get:5 https://mirrors.aliyun.com/kubernetes/apt kubernetes-xenia
l/main amd64 kubeadm amd64 1.10.1-00 [20.7 MB]
Fetched 56.9 MB in 5s (11.0 MB/s)
Use of uninitialized value $_ in lc at /usr/share/perl5/Debconf/
Template.pm line 287.
Selecting previously unselected package kubernetes-cni.
(Reading database ... 191799 files and directories currently ins
talled.)
Preparing to unpack .../kubernetes-cni_0.6.0-00_amd64.deb ...
Unpacking kubernetes-cni (0.6.0-00) ...
Selecting previously unselected package socat.
Preparing to unpack .../socat_1.7.3.1-1_amd64.deb ...
Unpacking ....
....
```

kubeadm安装完以后，就可以使用它来快速安装部署Kubernetes集群了。

使用kubeadm安装Kubernetes集群

使用kubeadmin初始化master节点

因为使用要使用canal，因此需要在初始化时加上网络配置参数,设置kubernetes的子网为10.244.0.0/16，注意此处不要修改为其他地址，因为这个值与后续的canal的yaml值要一致，如果修改，请一并修改。

这个下载镜像的过程涉及翻墙，因为会从gcr的站点下载容器镜像。。。（如果大家翻墙不方便的话，可以用我在上文准备工作中提到的导出的镜像）。

如果有能够连接gcr站点的网络，那么整个安装过程非常简单。

```
# kubeadm init --pod-network-cidr=10.244.0.0/16 --apiserver-adve
rtise-address=192.168.0.200
[init] Using Kubernetes version: v1.10.1
[init] Using Authorization modes: [Node RBAC]
[preflight] Running pre-flight checks.
    [WARNING FileExisting-crictl]: crictl not found in system pa
th
Suggestion: go get github.com/kubernetes-incubator/cri-tools/cmd
/crictl
[preflight] Starting the kubelet service
[certificates] Generated ca certificate and key.
[certificates] Generated apiserver certificate and key.
[certificates] apiserver serving cert is signed for DNS names [u
buntu-master kubernetes kubernetes.default kubernetes.default.sv
c kubernetes.default.svc.cluster.local] and IPs [10.96.0.1 192.1
68.0.200]
[certificates] Generated apiserver-kubelet-client certificate an
d key.
[certificates] Generated etcd/ca certificate and key.
[certificates] Generated etcd/server certificate and key.
[certificates] etcd/server serving cert is signed for DNS names [
localhost] and IPs [127.0.0.1]
[certificates] Generated etcd/peer certificate and key.
[certificates] etcd/peer serving cert is signed for DNS names [u
buntu-master] and IPs [192.168.0.200]
[certificates] Generated etcd/healthcheck-client certificate and
 key.
[certificates] Generated apiserver-etcd-client certificate and k
ey.
[certificates] Generated sa key and public key.
[certificates] Generated front-proxy-ca certificate and key.
[certificates] Generated front-proxy-client certificate and key.
[certificates] Valid certificates and keys now exist in "/etc/ku
bernetes/pki"
[kubeconfig] Wrote KubeConfig file to disk: "/etc/kubernetes/adm
in.conf"
[kubeconfig] Wrote KubeConfig file to disk: "/etc/kubernetes/kub
elet.conf"
[kubeconfig] Wrote KubeConfig file to disk: "/etc/kubernetes/con
troller-manager.conf"
[kubeconfig] Wrote KubeConfig file to disk: "/etc/kubernetes/sch
```

```
eduler.conf"
[controlplane] Wrote Static Pod manifest for component kube-apis
erver to "/etc/kubernetes/manifests/kube-apiserver.yaml"
[controlplane] Wrote Static Pod manifest for component kube-cont
roller-manager to "/etc/kubernetes/manifests/kube-controller-man
ager.yaml"
[controlplane] Wrote Static Pod manifest for component kube-sche
duler to "/etc/kubernetes/manifests/kube-scheduler.yaml"
[etcd] Wrote Static Pod manifest for a local etcd instance to "/
etc/kubernetes/manifests/etcd.yaml"
[init] Waiting for the kubelet to boot up the control plane as S
tatic Pods from directory "/etc/kubernetes/manifests".
[init] This might take a minute or longer if the control plane i
mages have to be pulled.
[apiclient] All control plane components are healthy after 28.00
3828 seconds
[uploadconfig] Storing the configuration used in ConfigMap "kube
adm-config" in the "kube-system" Namespace
[markmaster] Will mark node ubuntu-master as master by adding a
label and a taint
[markmaster] Master ubuntu-master tainted and labelled with key/
value: node-role.kubernetes.io/master=""
[bootstraptoken] Using token: rw4enn.mvk547juq7qi2b5f
[bootstraptoken] Configured RBAC rules to allow Node Bootstrap t
okens to post CSRs in order for nodes to get long term certifica
te credentials
[bootstraptoken] Configured RBAC rules to allow the csrapprover
controller automatically approve CSRs from a Node Bootstrap Toke
n
[bootstraptoken] Configured RBAC rules to allow certificate rota
tion for all node client certificates in the cluster
[bootstraptoken] Creating the "cluster-info" ConfigMap in the "k
ube-public" namespace
[addons] Applied essential addon: kube-dns
[addons] Applied essential addon: kube-proxy

Your Kubernetes master has initialized successfully!

To start using your cluster, you need to run the following as a
regular user:

  mkdir -p $HOME/.kube
  sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
  sudo chown $(id -u):$(id -g) $HOME/.kube/config

You should now deploy a pod network to the cluster.
Run "kubectl apply -f [podnetwork].yaml" with one of the options
 listed at:
  https://kubernetes.io/docs/concepts/cluster-administration/add
ons/

You can now join any number of machines by running the following
 on each node
```

```
as root:

  kubeadm join 192.168.0.200:6443 --token rw4enn.mvk547juq7qi2b5
f --discovery-token-ca-cert-hash sha256:ba260d5191213382a806a9a7
d92c9e6bb09061847c7914b1ac584d0c69471579
```

执行如下命令来配置kubectl。

```
mkdir -p $HOME/.kube
sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
sudo chown $(id -u):$(id -g) $HOME/.kube/config
```

这样master的节点就配置好了，并且可以使用kubectl来进行各种操作了，根据上面的提示接着往下做，将slave节点加入到集群。

Slave节点加入集群

在slave节点执行如下的命令,将slave节点加入集群，正常的返回信息如下：

```
#kubeadm join 192.168.0.200:6443 --token rw4enn.mvk547juq7qi2b5f
 --discovery-token-ca-cert-hash sha256:ba260d5191213382a806a9a7d
92c9e6bb09061847c7914b1ac584d0c69471579
[preflight] Running pre-flight checks.
    [WARNING FileExisting-crictl]: crictl not found in system pa
th
Suggestion: go get github.com/kubernetes-incubator/cri-tools/cmd
/crictl
[discovery] Trying to connect to API Server "192.168.0.200:6443"
[discovery] Created cluster-info discovery client, requesting in
fo from "https://192.168.0.200:6443"
[discovery] Requesting info from "https://192.168.0.200:6443" ag
ain to validate TLS against the pinned public key
[discovery] Cluster info signature and contents are valid and TL
S certificate validates against pinned roots, will use API Serve
r "192.168.0.200:6443"
[discovery] Successfully established connection with API Server
"192.168.0.200:6443"

This node has joined the cluster:
* Certificate signing request was sent to master and a response
  was received.
* The Kubelet was informed of the new secure connection details.

Run 'kubectl get nodes' on the master to see this node join the
cluster.
```

等待节点加入完毕。加入中状态。

```
# kubectl get node
NAME            STATUS     ROLES     AGE       VERSION
ubuntu-1        NotReady   <none>    6m        v1.10.1
ubuntu-2        NotReady   <none>    6m        v1.10.1
ubuntu-3        NotReady   <none>    6m        v1.10.1
ubuntu-master   NotReady   master    10m       v1.10.1
```

在master节点查看信息如下状态为节点加入完毕。

```
root@Ubuntu-master:~# kubectl get pod -n kube-system -o wide
NAME                                    READY     STATUS    REST
ARTS   AGE       IP              NODE
etcd-ubuntu-master                      1/1       Running   0
       21m       192.168.0.200   ubuntu-master
kube-apiserver-ubuntu-master            1/1       Running   0
       21m       192.168.0.200   ubuntu-master
kube-controller-manager-ubuntu-master   1/1       Running   0
       22m       192.168.0.200   ubuntu-master
kube-dns-86f4d74b45-wkfk2               0/3       Pending   0
       22m       <none>          <none>
kube-proxy-6ddb4                        1/1       Running   0
       22m       192.168.0.200   ubuntu-master
kube-proxy-7ngb9                        1/1       Running   0
       17m       192.168.0.202   ubuntu-2
kube-proxy-fkhhx                        1/1       Running   0
       18m       192.168.0.201   ubuntu-1
kube-proxy-rh4lq                        1/1       Running   0
       18m       192.168.0.203   ubuntu-3
kube-scheduler-ubuntu-master            1/1       Running   0
       21m       192.168.0.200   ubuntu-master
```

kubedns组件需要在网络插件完成安装以后会自动安装完成。

安装网络插件canal

从canal官方文档参考，如下网址下载2个文件并且安装，其中一个是配置canal的RBAC权限，一个是部署canal的DaemonSet。

```
# kubectl apply -f  https://docs.projectcalico.org/v3.0/getting-started/kubernetes/installation/hosted/canal/rbac.yaml
clusterrole.rbac.authorization.k8s.io "calico" created
clusterrole.rbac.authorization.k8s.io "flannel" created
clusterrolebinding.rbac.authorization.k8s.io "canal-flannel" created
clusterrolebinding.rbac.authorization.k8s.io "canal-calico" created


# kubectl apply -f https://docs.projectcalico.org/v3.0/getting-started/kubernetes/installation/hosted/canal/canal.yaml
configmap "canal-config" created
daemonset.extensions "canal" created
customresourcedefinition.apiextensions.k8s.io "felixconfigurations.crd.projectcalico.org" created
customresourcedefinition.apiextensions.k8s.io "bgpconfigurations.crd.projectcalico.org" created
customresourcedefinition.apiextensions.k8s.io "ippools.crd.projectcalico.org" created
customresourcedefinition.apiextensions.k8s.io "clusterinformations.crd.projectcalico.org" created
customresourcedefinition.apiextensions.k8s.io "globalnetworkpolicies.crd.projectcalico.org" created
customresourcedefinition.apiextensions.k8s.io "networkpolicies.crd.projectcalico.org" created
serviceaccount "canal" created
```

查看canal的安装状态。

```
# kubectl get pod -n kube-system -o wide
NAME                                    READY     STATUS    REST
ARTS   AGE       IP              NODE
canal-fc94k                             3/3       Running   10
       4m        192.168.0.201   ubuntu-1
canal-rs2wp                             3/3       Running   10
       4m        192.168.0.200   ubuntu-master
canal-tqd4l                             3/3       Running   10
       4m        192.168.0.202   ubuntu-2
canal-vmpnr                             3/3       Running   10
       4m        192.168.0.203   ubuntu-3
etcd-ubuntu-master                      1/1       Running   0
       28m       192.168.0.200   ubuntu-master
kube-apiserver-ubuntu-master            1/1       Running   0
       28m       192.168.0.200   ubuntu-master
kube-controller-manager-ubuntu-master   1/1       Running   0
       29m       192.168.0.200   ubuntu-master
kube-dns-86f4d74b45-wkfk2               3/3       Running   0
       28m       10.244.2.2      ubuntu-3
kube-proxy-6ddb4                        1/1       Running   0
       28m       192.168.0.200   ubuntu-master
kube-proxy-7ngb9                        1/1       Running   0
       24m       192.168.0.202   ubuntu-2
kube-proxy-fkhhx                        1/1       Running   0
       24m       192.168.0.201   ubuntu-1
kube-proxy-rh4lq                        1/1       Running   0
       24m       192.168.0.203   ubuntu-3
kube-scheduler-ubuntu-master            1/1       Running   0
       28m       192.168.0.200   ubuntu-master
```

可以看到canal和kube-dns都已经运行正常，一个基本功能正常的测试环境就部署完毕了。

此时查看集群的节点状态，版本为最新的版本v1.10.1。

```
# kubectl get node
NAME            STATUS    ROLES     AGE       VERSION
ubuntu-1        Ready     <none>    27m       v1.10.1
ubuntu-2        Ready     <none>    27m       v1.10.1
ubuntu-3        Ready     <none>    27m       v1.10.1
ubuntu-master   Ready     master    31m       v1.10.1
```

让master也运行pod（默认master不运行pod）,这样在测试环境做是可以的，不建议在生产环境如此操作。

```
#kubectl taint nodes --all node-role.kubernetes.io/master-
node "ubuntu-master" untainted
taint "node-role.kubernetes.io/master:" not found
taint "node-role.kubernetes.io/master:" not found
taint "node-role.kubernetes.io/master:" not found
```

后续如果想要集群其他功能启用，请参考后续文章。

服务发现与负载均衡

Kubernetes在设计之初就充分考虑了针对容器的服务发现与负载均衡机制，提供了Service资源，并通过kube-proxy配合cloud provider来适应不同的应用场景。随着kubernetes用户的激增，用户场景的不断丰富，又产生了一些新的负载均衡机制。目前，kubernetes中的负载均衡大致可以分为以下几种机制，每种机制都有其特定的应用场景：

- Service：直接用Service提供cluster内部的负载均衡，并借助cloud provider提供的LB提供外部访问
- Ingress Controller：还是用Service提供cluster内部的负载均衡，但是通过自定义LB提供外部访问
- Service Load Balancer：把load balancer直接跑在容器中，实现Bare Metal的Service Load Balancer
- Custom Load Balancer：自定义负载均衡，并替代kube-proxy，一般在物理部署Kubernetes时使用，方便接入公司已有的外部服务

Service

Service是对一组提供相同功能的Pods的抽象，并为它们提供一个统一的入口。借助Service，应用可以方便的实现服务发现与负载均衡，并实现应用的零宕机升级。Service通过标签来选取服务后端，一般配合Replication Controller或者Deployment来保证后端容器的正常运行。

Service有三种类型：

- ClusterIP：默认类型，自动分配一个仅cluster内部可以访问的虚拟IP
- NodePort：在ClusterIP基础上为Service在每台机器上绑定一个端口，这样就可以通过 `<NodeIP>:NodePort` 来访问该服务
- LoadBalancer：在NodePort的基础上，借助cloud provider创建一个外部的负载均衡器，并将请求转发到 `<NodeIP>:NodePort`

另外，也可以将已有的服务以Service的形式加入到Kubernetes集群中来，只需要在创建Service的时候不指定Label selector，而是在Service创建好后手动为其添加endpoint。

Ingress Controller

Service虽然解决了服务发现和负载均衡的问题，但它在使用上还是有一些限制，比如

- 对外访问的时候，NodePort类型需要在外部搭建额外的负载均衡，而LoadBalancer要求kubernetes必须跑在支持的cloud provider上面

Ingress就是为了解决这些限制而引入的新资源，主要用来将服务暴露到cluster外面，并且可以自定义服务的访问策略。比如想要通过负载均衡器实现不同子域名到不同服务的访问：

```
foo.bar.com --|                 |-> foo.bar.com s1:80
              | 178.91.123.132  |
bar.foo.com --|                 |-> bar.foo.com s2:80
```

可以这样来定义Ingress：

```yaml
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: test
spec:
  rules:
  - host: foo.bar.com
    http:
      paths:
      - backend:
          serviceName: s1
          servicePort: 80
  - host: bar.foo.com
    http:
      paths:
      - backend:
          serviceName: s2
          servicePort: 80
```

注意： Ingress本身并不会自动创建负载均衡器，cluster中需要运行一个ingress controller来根据Ingress的定义来管理负载均衡器。目前社区提供了nginx和gce的参考实现。

Traefik提供了易用的Ingress Controller，使用方法见https://docs.traefik.io/user-guide/kubernetes/ 。

Service Load Balancer

在Ingress出现以前，Service Load Balancer是推荐的解决Service局限性的方式。Service Load Balancer将haproxy跑在容器中，并监控service和endpoint的变化，通过容器IP对外提供4层和7层负载均衡服务。

社区提供的Service Load Balancer支持四种负载均衡协议：TCP、HTTP、HTTPS和SSL TERMINATION，并支持ACL访问控制。

Custom Load Balancer

虽然Kubernetes提供了丰富的负载均衡机制，但在实际使用的时候，还是会碰到一些复杂的场景是它不能支持的，比如：

- 接入已有的负载均衡设备
- 多租户网络情况下，容器网络和主机网络是隔离的，这样 `kube-proxy` 就不能正常工作

这个时候就可以自定义组件，并代替kube-proxy来做负载均衡。基本的思路是监控kubernetes中service和endpoints的变化，并根据这些变化来配置负载均衡器。比如weave flux、nginx plus、kube2haproxy等。

Endpoints

有几种情况下需要用到没有selector的service。

- 使用kubernetes集群外部的数据库时
- service中用到了其他namespace或kubernetes集群中的service
- 在kubernetes的工作负载与集群外的后端之间互相迁移

可以这样定义一个没有selector的service。

```
kind: Service
apiVersion: v1
metadata:
  name: my-service
spec:
  ports:
    - protocol: TCP
      port: 80
      targetPort: 9376
```

定义一个Endpoints来对应该service。

```
kind: Endpoints
apiVersion: v1
metadata:
  name: my-service
subsets:
  - addresses:
      - ip: 1.2.3.4
    ports:
      - port: 9376
```

访问没有selector的service跟访问有selector的service时没有任何区别。

使用kubernetes时有一个很常见的需求，就是当数据库部署在kubernetes集群之外的时候，集群内的service如何访问数据库呢？当然你可以直接使用数据库的IP地址和端口号来直接访问，有没有什么优雅的方式呢？你需要用到 `ExternalName Service` 。

```
kind: Service
apiVersion: v1
metadata:
  name: my-service
  namespace: prod
spec:
  type: ExternalName
  externalName: my.database.example.com
  ports:
  - port: 12345
```

这个例子中，在kubernetes集群内访问 `my-service` 实际上会重定向到 `my.database.example.com:12345` 这个地址。

参考资料

- https://kubernetes.io/docs/concepts/services-networking/service/
- http://kubernetes.io/docs/user-guide/ingress/
- https://github.com/kubernetes/contrib/tree/master/service-loadbalancer
- https://www.nginx.com/blog/load-balancing-kubernetes-services-nginx-plus/
- https://github.com/weaveworks/flux
- https://github.com/AdoHe/kube2haproxy

安装traefik ingress

Ingress简介

如果你还不了解，ingress是什么，可以先看下我翻译的Kubernetes官网上ingress的介绍Kubernetes Ingress解析。

理解Ingress

简单的说，ingress就是从kubernetes集群外访问集群的入口，将用户的URL请求转发到不同的service上。Ingress相当于nginx、apache等负载均衡方向代理服务器，其中还包括规则定义，即URL的路由信息，路由信息得的刷新由Ingress controller来提供。

理解Ingress Controller

Ingress Controller 实质上可以理解为是个监视器，Ingress Controller 通过不断地跟 kubernetes API 打交道，实时的感知后端 service、pod 等变化，比如新增和减少 pod，service 增加与减少等；当得到这些变化信息后，Ingress Controller 再结合下文的 Ingress 生成配置，然后更新反向代理负载均衡器，并刷新其配置，达到服务发现的作用。

部署Traefik

介绍traefik

Traefik是一款开源的反向代理与负载均衡工具。它最大的优点是能够与常见的微服务系统直接整合，可以实现自动化动态配置。目前支持Docker, Swarm, Mesos/Marathon, Mesos, Kubernetes, Consul, Etcd, Zookeeper, BoltDB, Rest API等等后端模型。

以下配置文件可以在kubernetes-handbookGitHub仓库中的../manifests/traefik-ingress/目录下找到。

创建ingress-rbac.yaml

将用于service account验证。

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: ingress
  namespace: kube-system

---

kind: ClusterRoleBinding
apiVersion: rbac.authorization.k8s.io/v1beta1
metadata:
  name: ingress
subjects:
  - kind: ServiceAccount
    name: ingress
    namespace: kube-system
roleRef:
  kind: ClusterRole
  name: cluster-admin
  apiGroup: rbac.authorization.k8s.io
```

创建名为 **`traefik-ingress`** 的**ingress**，文件名ingress.yaml

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: traefik-ingress
  namespace: default
spec:
  rules:
  - host: traefik.nginx.io
    http:
      paths:
      - path: /
        backend:
          serviceName: my-nginx
          servicePort: 80
  - host: traefik.frontend.io
    http:
      paths:
      - path: /
        backend:
          serviceName: frontend
          servicePort: 80
```

这其中的 `backend` 中要配置default namespace中启动的service名字，如果你没有配置namespace名字，默认使用default namespace，如果你在其他namespace中创建服务想要暴露到kubernetes集群外部，可以创建新的ingress.yaml文件，同

时在文件中指定该 `namespace` ，其他配置与上面的文件格式相同。。 `path` 就是URL地址后的路径，如traefik.frontend.io/path，service将会接受path这个路径，host最好使用service-name.filed1.filed2.domain-name这种类似主机名称的命名方式，方便区分服务。

根据你自己环境中部署的service的名字和端口自行修改，有新service增加时，修改该文件后可以使用 `kubectl replace -f ingress.yaml` 来更新。

我们现在集群中已经有两个service了，一个是nginx，另一个是官方的 `guestbook` 例子。

创建**DaemonSet**

我们使用DaemonSet类型来部署Traefik，并使用 `nodeSelector` 来限定Traefik所部署的主机。

```
apiVersion: extensions/v1beta1
kind: DaemonSet
metadata:
  name: traefik-ingress-lb
  namespace: kube-system
  labels:
    k8s-app: traefik-ingress-lb
spec:
  template:
    metadata:
      labels:
        k8s-app: traefik-ingress-lb
        name: traefik-ingress-lb
    spec:
      terminationGracePeriodSeconds: 60
      hostNetwork: true
      restartPolicy: Always
      serviceAccountName: ingress
      containers:
      - image: traefik
        name: traefik-ingress-lb
        resources:
          limits:
            cpu: 200m
            memory: 30Mi
          requests:
            cpu: 100m
            memory: 20Mi
        ports:
        - name: http
          containerPort: 80
          hostPort: 80
        - name: admin
          containerPort: 8580
          hostPort: 8580
        args:
        - --web
        - --web.address=:8580
        - --kubernetes
      nodeSelector:
        edgenode: "true"
```

注意：我们使用了 `nodeSelector` 选择边缘节点来调度traefik-ingress-lb运行在它上面，所有你需要使用：

```
kubectl label nodes 172.20.0.113 edgenode=true
kubectl label nodes 172.20.0.114 edgenode=true
kubectl label nodes 172.20.0.115 edgenode=true
```

给三个node打标签，这样traefik的pod才会调度到这几台主机上，否则会一直处于 `pending` 状态。

关于使用Traefik作为边缘节点请参考边缘节点配置。

Traefik UI

使用下面的yaml配置来创建Traefik的Web UI。

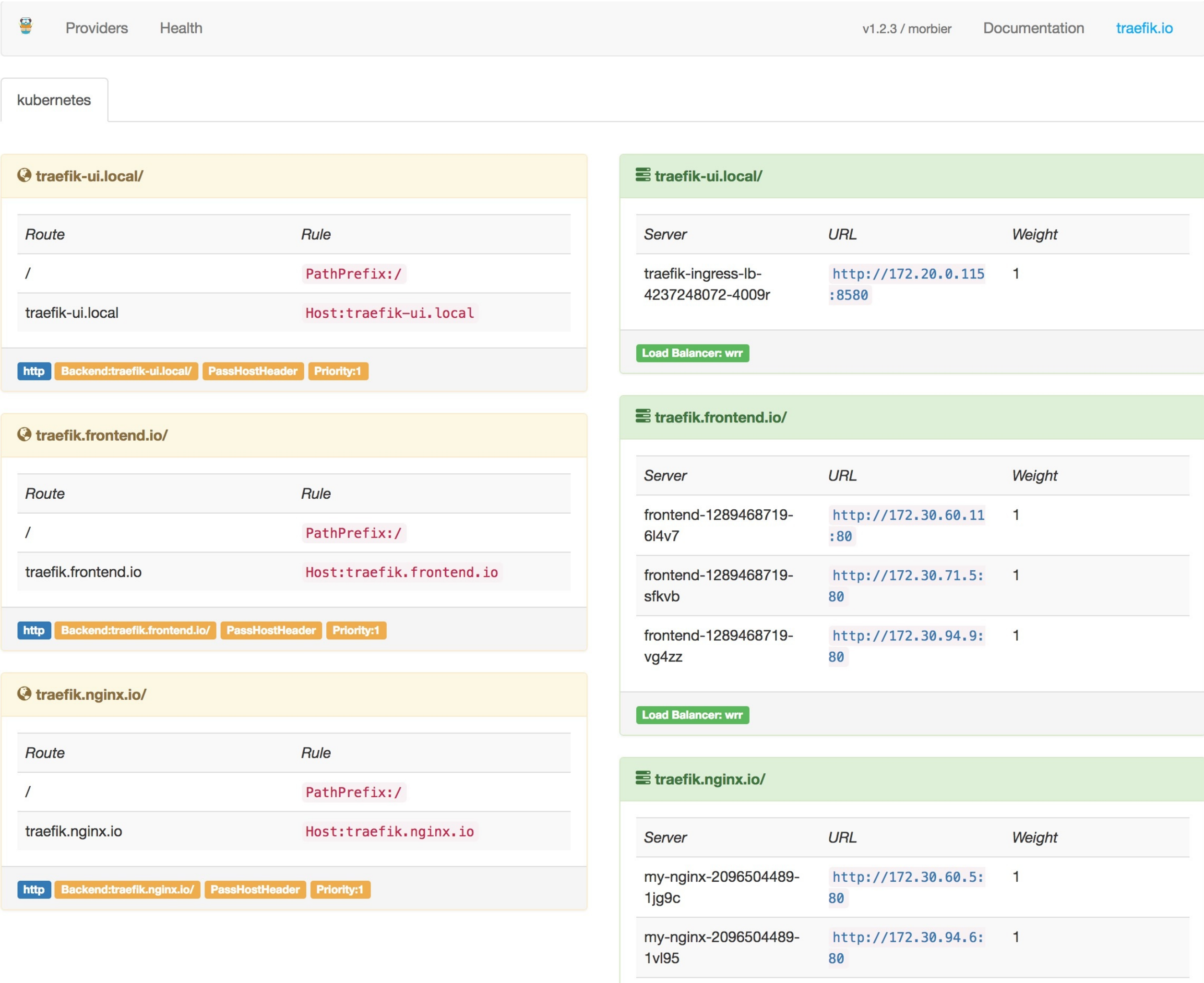
```yaml
apiVersion: v1
kind: Service
metadata:
  name: traefik-web-ui
  namespace: kube-system
spec:
  selector:
    k8s-app: traefik-ingress-lb
  ports:
  - name: web
    port: 80
    targetPort: 8580
---
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: traefik-web-ui
  namespace: kube-system
spec:
  rules:
  - host: traefik-ui.local
    http:
      paths:
      - path: /
        backend:
          serviceName: traefik-web-ui
          servicePort: web
```

配置完成后就可以启动treafik ingress了。

```
kubectl create -f .
```

我查看到traefik的pod在 `172.20.0.115` 这台节点上启动了。

访问该地址 `http://172.20.0.115:8580/` 将可以看到dashboard。



图片 - *kubernetes-dashboard*

左侧黄色部分部分列出的是所有的rule，右侧绿色部分是所有的backend。

测试

在集群的任意一个节点上执行。假如现在我要访问nginx的"/"路径。

```
$ curl -H Host:traefik.nginx.io http://172.20.0.115/
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
    body {
        width: 35em;
        margin: 0 auto;
        font-family: Tahoma, Verdana, Arial, sans-serif;
    }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully in
stalled and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
```

如果你需要在kubernetes集群以外访问就需要设置DNS，或者修改本机的hosts文件。

在其中加入：

```
172.20.0.115 traefik.nginx.io
172.20.0.115 traefik.frontend.io
```

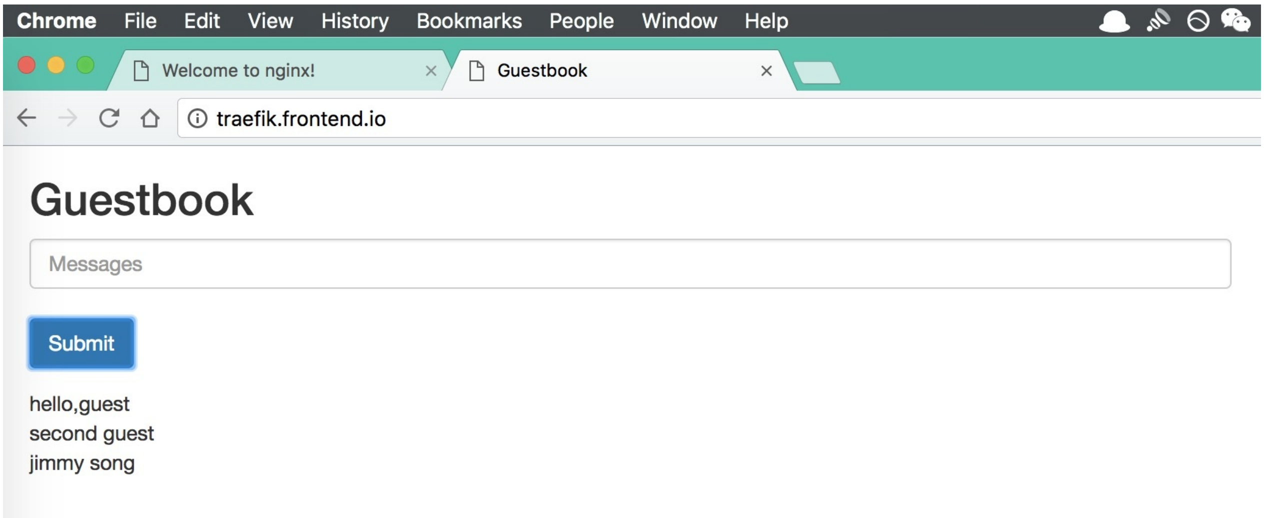
所有访问这些地址的流量都会发送给172.20.0.115这台主机，就是我们启动traefik的主机。

Traefik会解析http请求header里的Host参数将流量转发给Ingress配置里的相应service。

修改hosts后就就可以在kubernetes集群外访问以上两个service，如下图：



图片 - *traefik-nginx*



图片 - *traefik-guestbook*

参考

- Traefik-kubernetes 初试
- Traefik简介
- Guestbook example

分布式负载测试

该教程描述如何在Kubernetes中进行分布式负载均衡测试，包括一个web应用、docker镜像和Kubernetes controllers/services。关于分布式负载测试的更多资料请查看Distributed Load Testing Using Kubernetes 。

准备

不需要**GCE**及其他组件，你只需要有一个**kubernetes**集群即可。

如果你还没有kubernetes集群，可以参考kubernetes-handbook部署一个。

部署Web应用

本文中使用的镜像、kubernetes应用的yaml配置来自我的另一个项目，请参考：https://github.com/rootsongjc/distributed-load-testing-using-kubernetes

`sample-webapp` 目录下包含一个简单的web测试应用。我们将其构建为docker镜像，在kubernetes中运行。你可以自己构建，也可以直接用这个我构建好的镜像 `index.tenxcloud.com/jimmy/k8s-sample-webapp:latest` 。

在kubernetes上部署sample-webapp。

```
$ git clone https://github.com/rootsongjc/distributed-load-testi
ng-using-kubernetes.git
$ cd kubernetes-config
$ kubectl create -f sample-webapp-controller.yaml
$ kubectl create -f sample-webapp-service.yaml
```

部署Locust的Controller和Service

`locust-master` 和 `locust-work` 使用同样的docker镜像，修改cotnroller中 `spec.template.spec.containers.env` 字段中的value为你 `sample-webapp` service的名字。

```
- name: TARGET_HOST
  value: http://sample-webapp:8000
```

创建Controller Docker镜像（可选）

`locust-master` 和 `locust-work` controller使用的都是 `locust-tasks` docker镜像。你可以直接下载 `gcr.io/cloud-solutions-images/locust-tasks` ，也可以自己编译。自己编译大概要花几分钟时间，镜像大小为820M。

```
$ docker build -t index.tenxcloud.com/jimmy/locust-tasks:latest
.
$ docker push index.tenxcloud.com/jimmy/locust-tasks:latest
```

注意：我使用的是时速云的镜像仓库。

每个controller的yaml的 `spec.template.spec.containers.image` 字段指定的是我的镜像：

```
image: index.tenxcloud.com/jimmy/locust-tasks:latest
```

部署locust-master

```
$ kubectl create -f locust-master-controller.yaml
$ kubectl create -f locust-master-service.yaml
```

部署locust-worker

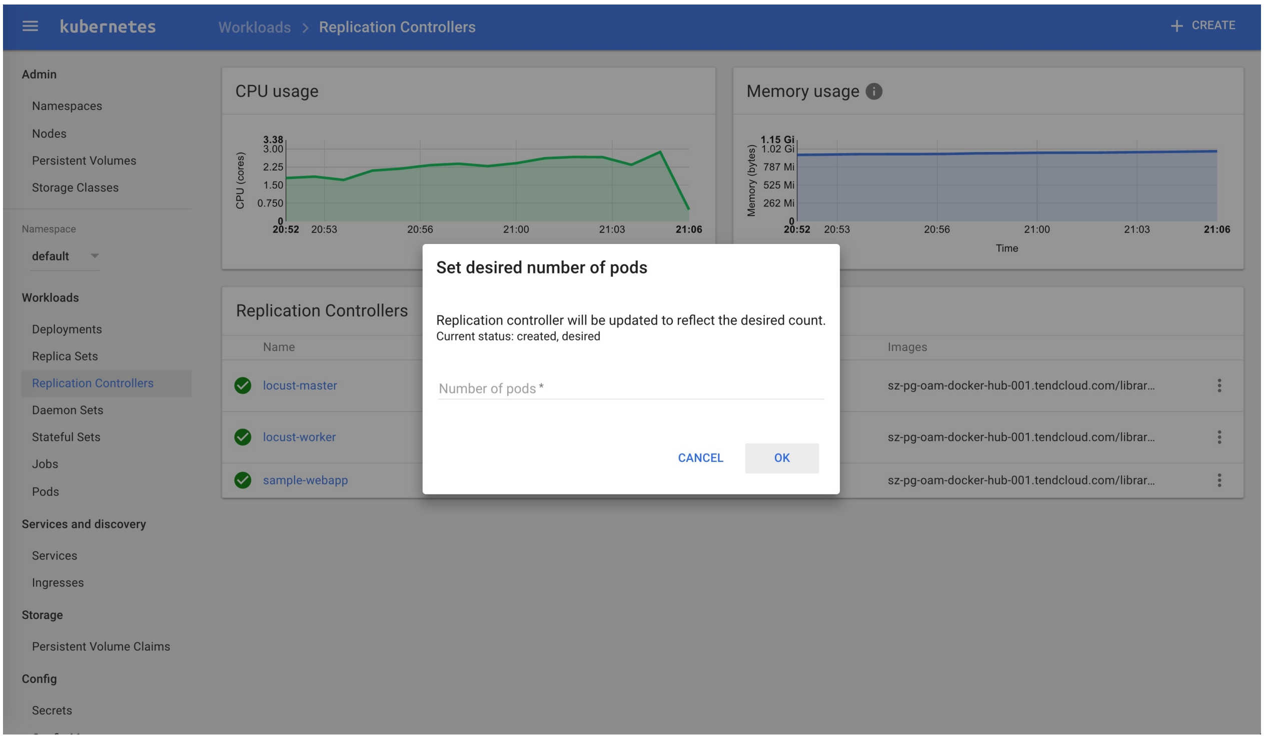
Now deploy `locust-worker-controller` :

```
$ kubectl create -f locust-worker-controller.yaml
```

你可以很轻易的给work扩容，通过命令行方式：

```
$ kubectl scale --replicas=20 replicationcontrollers locust-work
er
```

当然你也可以通过WebUI：Dashboard - Workloads - Replication Controllers - **ServiceName** - Scale来扩容。



图片 - 使用*dashboard*来扩容

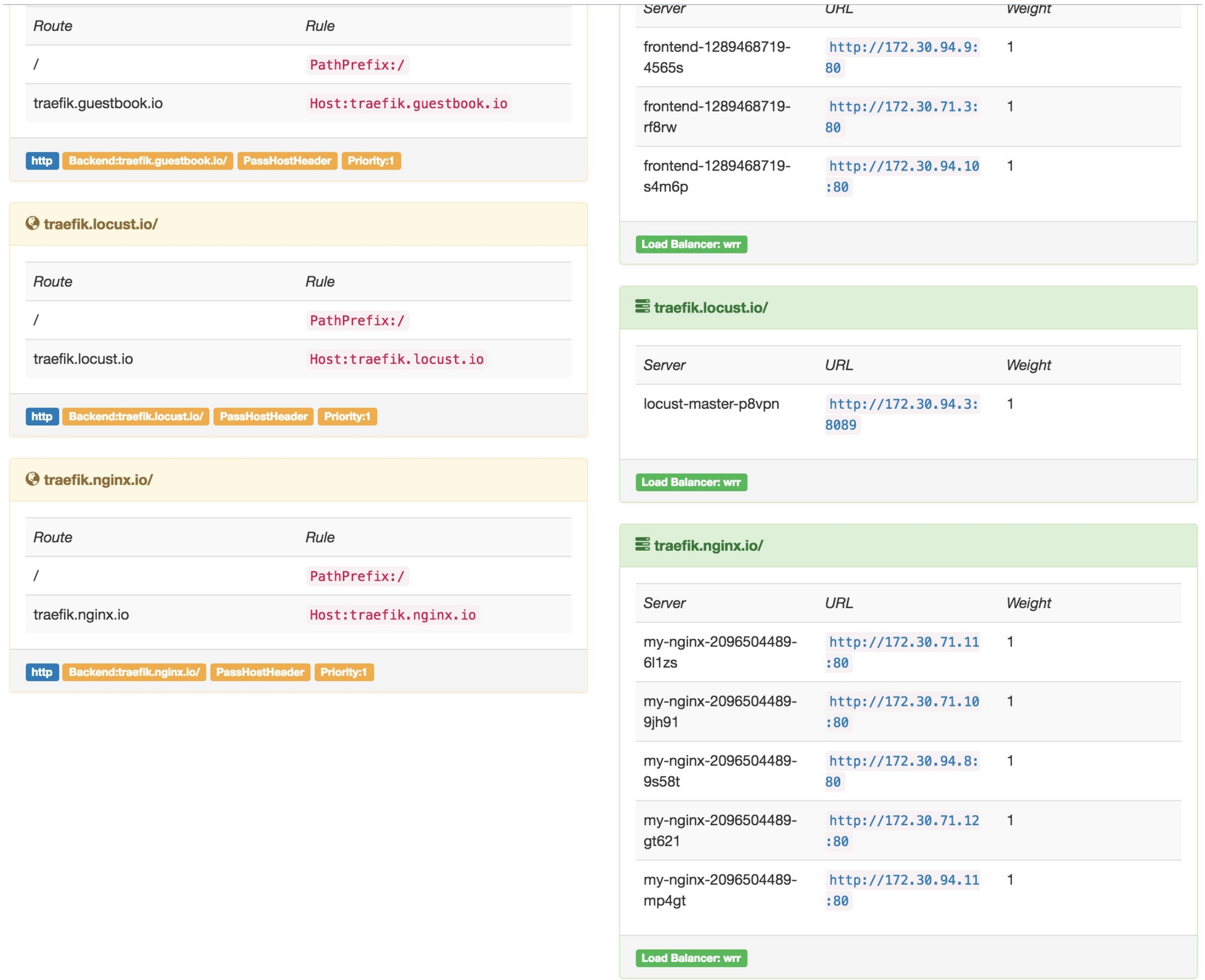
配置Traefik

参考kubernetes的traefik ingress安装，在 `ingress.yaml` 中加入如下配置：

```
  - host: traefik.locust.io
    http:
      paths:
      - path: /
        backend:
          serviceName: locust-master
          servicePort: 8089
```

然后执行 `kubectl replace -f ingress.yaml` 即可更新traefik。

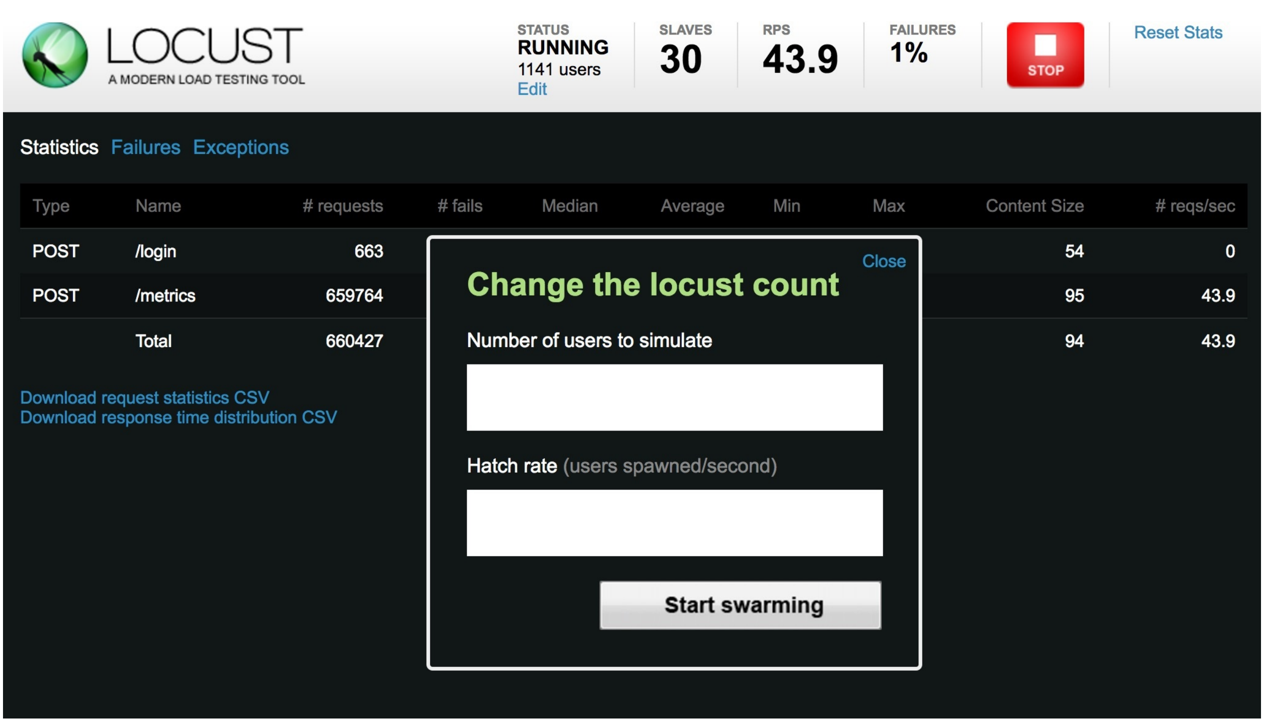
通过Traefik的dashboard就可以看到刚增加的 `traefik.locust.io` 节点。



图片 - *Traefik的UI*

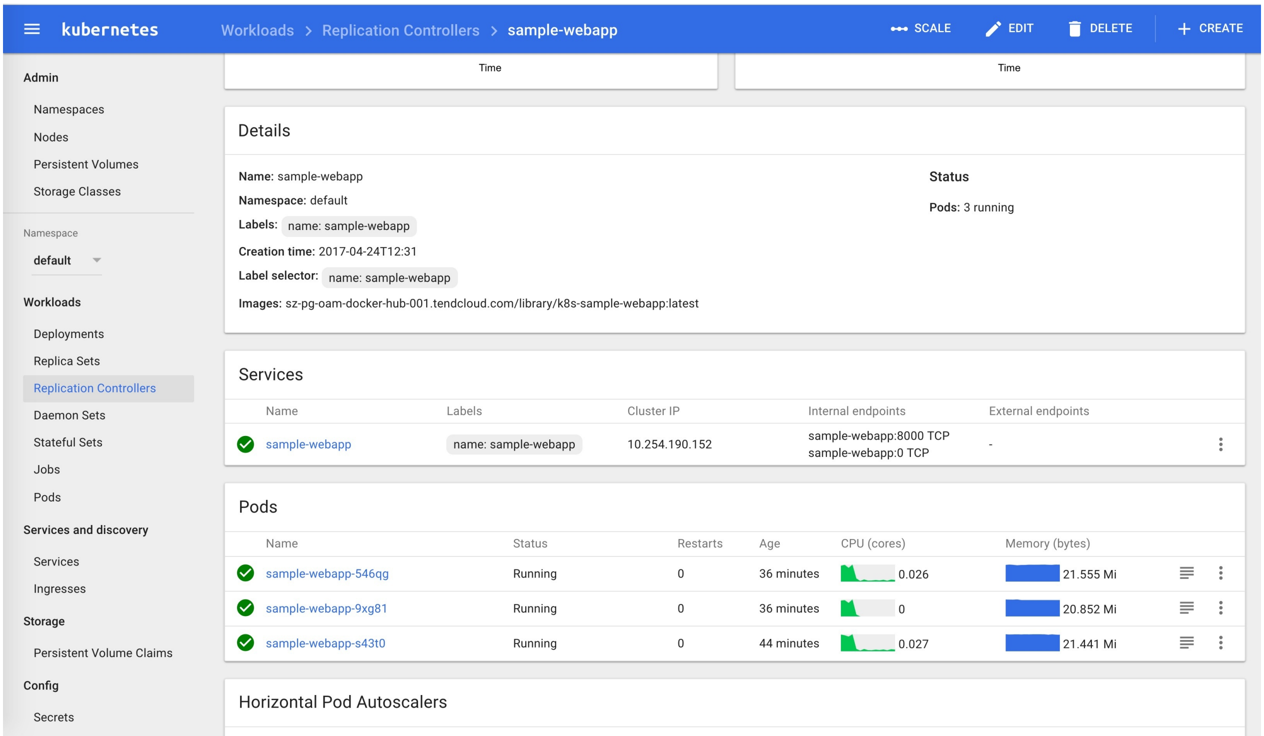
执行测试

打开 `http://traefik.locust.io` 页面，点击 `Edit` 输入伪造的用户数和用户每秒发送的请求个数，点击 `Start Swarming` 就可以开始测试了。



图片 - *Locust*启动界面

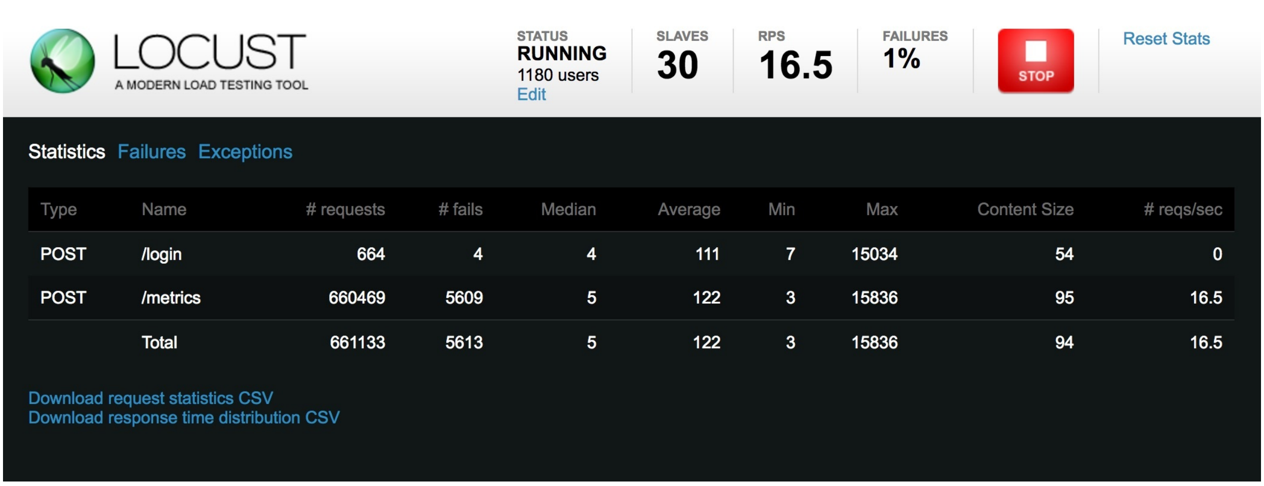
在测试过程中调整 `sample-webapp` 的pod个数（默认设置了1个pod），观察pod的负载变化情况。



图片 - *Dashboard*查看页面

从一段时间的观察中可以看到负载被平均分配给了3个pod。

在locust的页面中可以实时观察也可以下载测试结果。



图片 - *Locust*测试结果页面

Kubernetes网络和集群性能测试

准备

测试环境

在以下几种环境下进行测试：

- Kubernetes集群node节点上通过Cluster IP方式访问
- Kubernetes集群内部通过service访问
- Kubernetes集群外部通过traefik ingress暴露的地址访问

测试地址

Cluster IP: 10.254.149.31

Service Port：8000

Ingress Host：traefik.sample-webapp.io

测试工具

- Locust：一个简单易用的用户负载测试工具，用来测试web或其他系统能够同时处理的并发用户数。
- curl
- kubemark
- 测试程序：sample-webapp，源码见Github kubernetes的分布式负载测试

测试说明

通过向 `sample-webapp` 发送curl请求获取响应时间，直接curl后的结果为：

```
$ curl "http://10.254.149.31:8000/"
Welcome to the "Distributed Load Testing Using Kubernetes" sampl
e web app
```

网络延迟测试

场景一、 Kubernetes集群node节点上通过Cluster IP访问

测试命令

```
curl -o /dev/null -s -w '%{time_connect} %{time_starttransfer} %{time_total}' "http://10.254.149.31:8000/"
```

10组测试结果

No	time_connect	time_starttransfer	time_total
1	0.000	0.003	0.003
2	0.000	0.002	0.002
3	0.000	0.002	0.002
4	0.000	0.002	0.002
5	0.000	0.002	0.002
6	0.000	0.002	0.002
7	0.000	0.002	0.002
8	0.000	0.002	0.002
9	0.000	0.002	0.002
10	0.000	0.002	0.002

平均响应时间：**2ms**

时间指标说明

单位：秒

time_connect：建立到服务器的 TCP 连接所用的时间

time_starttransfer：在发出请求之后，Web 服务器返回数据的第一个字节所用的时间

time_total：完成请求所用的时间

场景二、Kubernetes集群内部通过service访问

测试命令

```
curl -o /dev/null -s -w '%{time_connect} %{time_starttransfer} %{time_total}' "http://sample-webapp:8000/"
```

10组测试结果

No	time_connect	time_starttransfer	time_total
1	0.004	0.006	0.006
2	0.004	0.006	0.006
3	0.004	0.006	0.006
4	0.004	0.006	0.006
5	0.004	0.006	0.006
6	0.004	0.006	0.006
7	0.004	0.006	0.006
8	0.004	0.006	0.006
9	0.004	0.006	0.006
10	0.004	0.006	0.006

平均响应时间：**6ms**

场景三、在公网上通过traefik ingress访问

测试命令

```
curl -o /dev/null -s -w '%{time_connect} %{time_starttransfer} %{time_total}' "http://traefik.sample-webapp.io" >>result
```

10组测试结果

No	time_connect	time_starttransfer	time_total
1	0.043	0.085	0.085
2	0.052	0.093	0.093
3	0.043	0.082	0.082
4	0.051	0.093	0.093
5	0.068	0.188	0.188
6	0.049	0.089	0.089
7	0.051	0.113	0.113
8	0.055	0.120	0.120
9	0.065	0.126	0.127
10	0.050	0.111	0.111

平均响应时间：**110ms**

测试结果

在这三种场景下的响应时间测试结果如下：

- Kubernetes集群node节点上通过Cluster IP方式访问：2ms
- Kubernetes集群内部通过service访问：6ms
- Kubernetes集群外部通过traefik ingress暴露的地址访问：110ms

注意：执行测试的*node*节点*/Pod*与*serivce*所在的*pod*的距离（是否在同一台主机上），对前两个场景可以能会有一定影响。

网络性能测试

网络使用flannel的vxlan模式。

使用iperf进行测试。

服务端命令：

```
iperf -s -p 12345 -i 1 -M
```

客户端命令：

```
iperf -c ${server-ip} -p 12345 -i 1 -t 10 -w 20K
```

场景一、主机之间

```
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0- 1.0 sec   598 MBytes  5.02 Gbits/sec
[  3]  1.0- 2.0 sec   637 MBytes  5.35 Gbits/sec
[  3]  2.0- 3.0 sec   664 MBytes  5.57 Gbits/sec
[  3]  3.0- 4.0 sec   657 MBytes  5.51 Gbits/sec
[  3]  4.0- 5.0 sec   641 MBytes  5.38 Gbits/sec
[  3]  5.0- 6.0 sec   639 MBytes  5.36 Gbits/sec
[  3]  6.0- 7.0 sec   628 MBytes  5.26 Gbits/sec
[  3]  7.0- 8.0 sec   649 MBytes  5.44 Gbits/sec
[  3]  8.0- 9.0 sec   638 MBytes  5.35 Gbits/sec
[  3]  9.0-10.0 sec   652 MBytes  5.47 Gbits/sec
[  3]  0.0-10.0 sec  6.25 GBytes  5.37 Gbits/sec
```

场景二、不同主机的Pod之间(使用flannel的vxlan模式)

```
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0- 1.0 sec   372 MBytes  3.12 Gbits/sec
[  3]  1.0- 2.0 sec   345 MBytes  2.89 Gbits/sec
[  3]  2.0- 3.0 sec   361 MBytes  3.03 Gbits/sec
[  3]  3.0- 4.0 sec   397 MBytes  3.33 Gbits/sec
[  3]  4.0- 5.0 sec   405 MBytes  3.40 Gbits/sec
[  3]  5.0- 6.0 sec   410 MBytes  3.44 Gbits/sec
[  3]  6.0- 7.0 sec   404 MBytes  3.39 Gbits/sec
[  3]  7.0- 8.0 sec   408 MBytes  3.42 Gbits/sec
[  3]  8.0- 9.0 sec   451 MBytes  3.78 Gbits/sec
[  3]  9.0-10.0 sec   387 MBytes  3.25 Gbits/sec
[  3]  0.0-10.0 sec  3.85 GBytes  3.30 Gbits/sec
```

场景三、Node与非同主机的Pod之间（使用flannel的vxlan模式）

```
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0- 1.0 sec   372 MBytes  3.12 Gbits/sec
[  3]  1.0- 2.0 sec   420 MBytes  3.53 Gbits/sec
[  3]  2.0- 3.0 sec   434 MBytes  3.64 Gbits/sec
[  3]  3.0- 4.0 sec   409 MBytes  3.43 Gbits/sec
[  3]  4.0- 5.0 sec   382 MBytes  3.21 Gbits/sec
[  3]  5.0- 6.0 sec   408 MBytes  3.42 Gbits/sec
[  3]  6.0- 7.0 sec   403 MBytes  3.38 Gbits/sec
[  3]  7.0- 8.0 sec   423 MBytes  3.55 Gbits/sec
[  3]  8.0- 9.0 sec   376 MBytes  3.15 Gbits/sec
[  3]  9.0-10.0 sec   451 MBytes  3.78 Gbits/sec
[  3]  0.0-10.0 sec  3.98 GBytes  3.42 Gbits/sec
```

场景四、不同主机的Pod之间（使用flannel的host-gw模式）

```
[ ID] Interval       Transfer     Bandwidth
[  5]  0.0- 1.0 sec   530 MBytes  4.45 Gbits/sec
[  5]  1.0- 2.0 sec   576 MBytes  4.84 Gbits/sec
[  5]  2.0- 3.0 sec   631 MBytes  5.29 Gbits/sec
[  5]  3.0- 4.0 sec   580 MBytes  4.87 Gbits/sec
[  5]  4.0- 5.0 sec   627 MBytes  5.26 Gbits/sec
[  5]  5.0- 6.0 sec   578 MBytes  4.85 Gbits/sec
[  5]  6.0- 7.0 sec   584 MBytes  4.90 Gbits/sec
[  5]  7.0- 8.0 sec   571 MBytes  4.79 Gbits/sec
[  5]  8.0- 9.0 sec   564 MBytes  4.73 Gbits/sec
[  5]  9.0-10.0 sec   572 MBytes  4.80 Gbits/sec
[  5]  0.0-10.0 sec  5.68 GBytes  4.88 Gbits/sec
```

场景五、Node与非同主机的Pod之间（使用flannel的host-gw模式）

```
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0- 1.0 sec   570 MBytes  4.78 Gbits/sec
[  3]  1.0- 2.0 sec   552 MBytes  4.63 Gbits/sec
[  3]  2.0- 3.0 sec   598 MBytes  5.02 Gbits/sec
[  3]  3.0- 4.0 sec   580 MBytes  4.87 Gbits/sec
[  3]  4.0- 5.0 sec   590 MBytes  4.95 Gbits/sec
[  3]  5.0- 6.0 sec   594 MBytes  4.98 Gbits/sec
[  3]  6.0- 7.0 sec   598 MBytes  5.02 Gbits/sec
[  3]  7.0- 8.0 sec   606 MBytes  5.08 Gbits/sec
[  3]  8.0- 9.0 sec   596 MBytes  5.00 Gbits/sec
[  3]  9.0-10.0 sec   604 MBytes  5.07 Gbits/sec
[  3]  0.0-10.0 sec  5.75 GBytes  4.94 Gbits/sec
```

网络性能对比综述

使用Flannel的**vxlan**模式实现每个pod一个IP的方式，会比宿主机直接互联的网络性能损耗30%～40%，符合网上流传的测试结论。而flannel的host-gw模式比起宿主机互连的网络性能损耗大约是10%。

Vxlan会有一个封包解包的过程，所以会对网络性能造成较大的损耗，而host-gw模式是直接使用路由信息，网络损耗小，关于host-gw的架构请访问Flannel host-gw architecture。

Kubernete的性能测试

参考Kubernetes集群性能测试中的步骤，对kubernetes的性能进行测试。

我的集群版本是Kubernetes1.6.0，首先克隆代码，将kubernetes目录复制到 `$GOPATH/src/k8s.io/` 下然后执行：

```
$ ./hack/generate-bindata.sh
/usr/local/src/k8s.io/kubernetes /usr/local/src/k8s.io/kubernete
s
Generated bindata file : test/e2e/generated/bindata.go has 13498
 test/e2e/generated/bindata.go lines of lovely automated artifac
ts
No changes in generated bindata file: pkg/generated/bindata.go
/usr/local/src/k8s.io/kubernetes
$ make WHAT="test/e2e/e2e.test"
...
+++ [0425 17:01:34] Generating bindata:
    test/e2e/generated/gobindata_util.go
/usr/local/src/k8s.io/kubernetes /usr/local/src/k8s.io/kubernete
s/test/e2e/generated
/usr/local/src/k8s.io/kubernetes/test/e2e/generated
+++ [0425 17:01:34] Building go targets for linux/amd64:
    test/e2e/e2e.test
$ make ginkgo
+++ [0425 17:05:57] Building the toolchain targets:
    k8s.io/kubernetes/hack/cmd/teststale
    k8s.io/kubernetes/vendor/github.com/jteeuwen/go-bindata/go-b
indata
+++ [0425 17:05:57] Generating bindata:
    test/e2e/generated/gobindata_util.go
/usr/local/src/k8s.io/kubernetes /usr/local/src/k8s.io/kubernete
s/test/e2e/generated
/usr/local/src/k8s.io/kubernetes/test/e2e/generated
+++ [0425 17:05:58] Building go targets for linux/amd64:
    vendor/github.com/onsi/ginkgo/ginkgo

$ export KUBERNETES_PROVIDER=local
$ export KUBECTL_PATH=/usr/bin/kubectl
$ go run hack/e2e.go -v -test  --test_args="--host=http://172.20
.0.113:8080 --ginkgo.focus=\[Feature:Performance\]" >>log.txt
```

测试结果

```
Apr 25 18:27:31.461: INFO: API calls latencies: {
  "apicalls": [
    {
      "resource": "pods",
      "verb": "POST",
      "latency": {
        "Perc50": 2148000,
        "Perc90": 13772000,
        "Perc99": 14436000,
        "Perc100": 0
      }
    },
    {
      "resource": "services",
      "verb": "DELETE",
      "latency": {
        "Perc50": 9843000,
        "Perc90": 11226000,
        "Perc99": 12391000,
        "Perc100": 0
      }
    },
    ...
Apr 25 18:27:31.461: INFO: [Result:Performance] {
  "version": "v1",
  "dataItems": [
    {
      "data": {
        "Perc50": 2.148,
        "Perc90": 13.772,
        "Perc99": 14.436
      },
      "unit": "ms",
      "labels": {
        "Resource": "pods",
        "Verb": "POST"
      }
    },
...
2.857: INFO: Running AfterSuite actions on all node
Apr 26 10:35:32.857: INFO: Running AfterSuite actions on node 1

Ran 2 of 606 Specs in 268.371 seconds
SUCCESS! -- 2 Passed | 0 Failed | 0 Pending | 604 Skipped PASS

Ginkgo ran 1 suite in 4m28.667870101s
Test Suite Passed
```

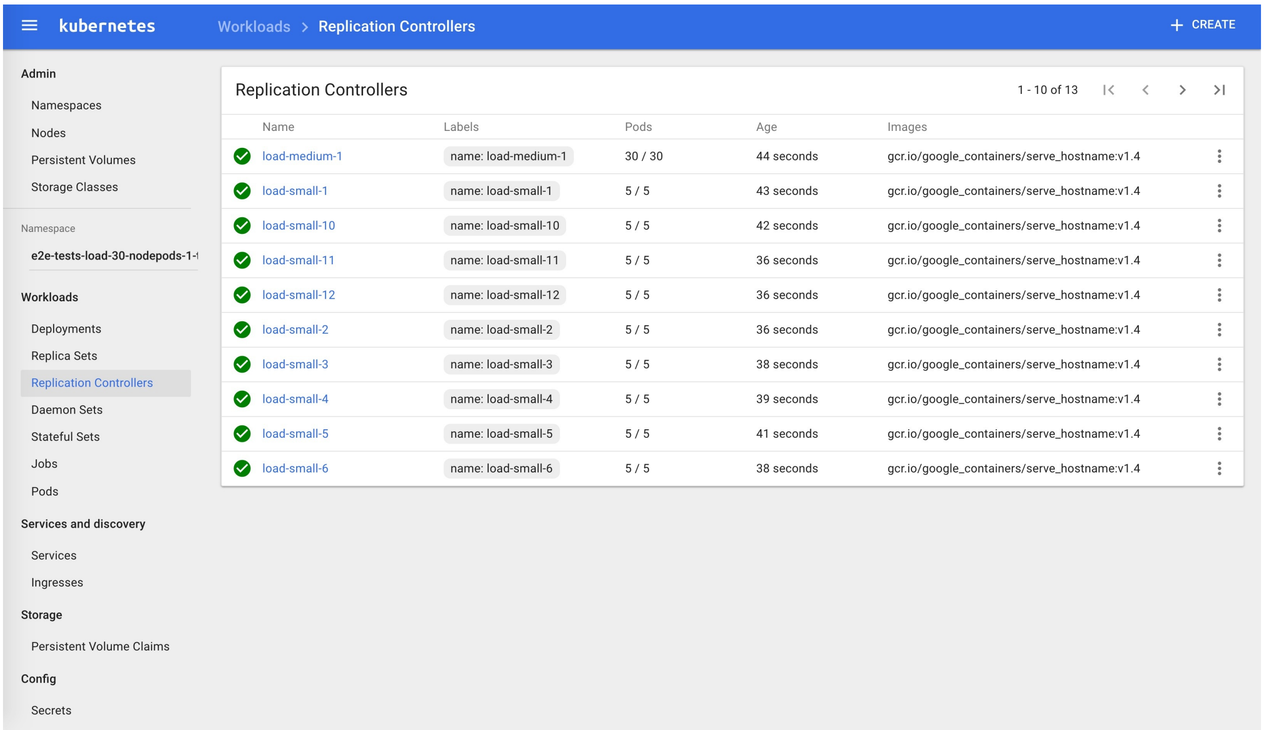
从kubemark输出的日志中可以看到**API calls latencies**和**Performance**。

日志里显示，创建**90**个**pod**用时**40**秒以内，平均创建每个**pod**耗时**0.44**秒。

不同type的资源类型API请求耗时分布

Resource	Verb	50%	90%	99%
services	DELETE	8.472ms	9.841ms	38.226ms
endpoints	PUT	1.641ms	3.161ms	30.715ms
endpoints	GET	931µs	10.412ms	27.97ms
nodes	PATCH	4.245ms	11.117ms	18.63ms
pods	PUT	2.193ms	2.619ms	17.285ms

从 `log.txt` 日志中还可以看到更多详细请求的测试指标。



图片 - *kubernetes-dashboard*

注意事项

测试过程中需要用到docker镜像存储在GCE中，需要翻墙下载，我没看到哪里配置这个镜像的地址。该镜像副本已上传时速云：

用到的镜像有如下两个：

- gcr.io/google_containers/pause-amd64:3.0
- gcr.io/google_containers/serve_hostname:v1.4

时速云镜像地址：

- index.tenxcloud.com/jimmy/pause-amd64:3.0
- index.tenxcloud.com/jimmy/serve_hostname:v1.4

将镜像pull到本地后重新打tag。

Locust测试

请求统计

Method	Name	# requests	# failures	Median response time	Average response time	Min respor time
POST	/login	5070	78	59000	80551	11218
POST	/metrics	5114232	85879	63000	82280	29518
None	Total	5119302	85957	63000	82279	11218

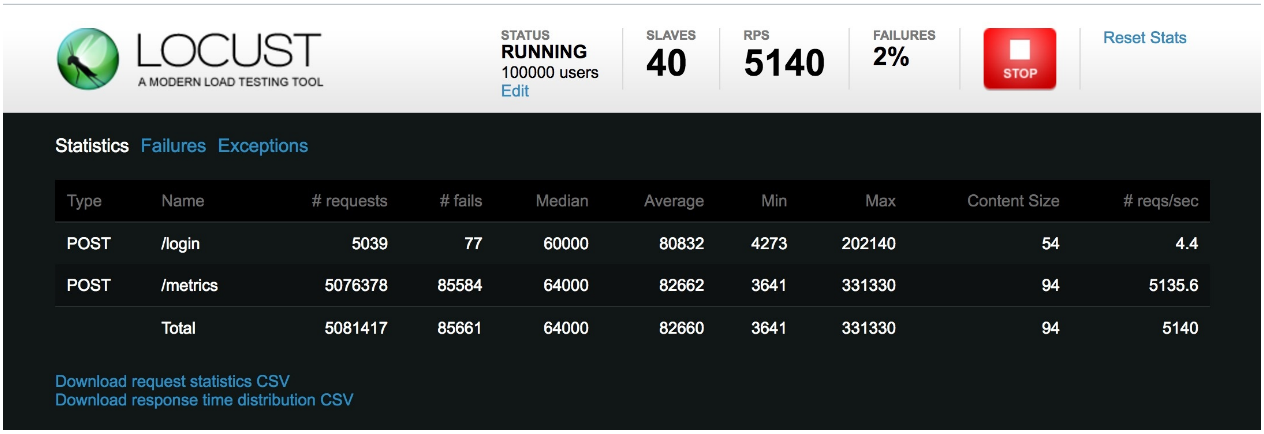
响应时间分布

Name	# requests	50%	66%	75%	80%	90%	9
POST /login	5070	59000	125000	140000	148000	160000	16
POST /metrics	5114993	63000	127000	142000	149000	160000	16
None Total	5120063	63000	127000	142000	149000	160000	16

以上两个表格都是瞬时值。请求失败率在2%左右。

Sample-webapp起了48个pod。

Locust模拟10万用户，每秒增长100个。



图片 - *locust*测试页面

关于Locust的使用请参考Github：https://github.com/rootsongjc/distributed-load-testing-using-kubernetes

参考

- 基于 Python 的性能测试工具 locust (与 LR 的简单对比)
- Locust docs
- python用户负载测试工具：locust
- Kubernetes集群性能测试
- CoreOS是如何将Kubernetes的性能提高10倍的
- Kubernetes 1.3 的性能和弹性 —— 2000 节点，60,0000 Pod 的集群
- 运用Kubernetes进行分布式负载测试
- Kubemark User Guide
- Flannel host-gw architecture

边缘节点配置

前言

为了配置kubernetes中的traefik ingress的高可用，对于kubernetes集群以外只暴露一个访问入口，需要使用keepalived排除单点问题。本文参考了kube-keepalived-vip，但并没有使用容器方式安装，而是直接在node节点上安装。

定义

首先解释下什么叫边缘节点（Edge Node），所谓的边缘节点即集群内部用来向集群外暴露服务能力的节点，集群外部的服务通过该节点来调用集群内部的服务，边缘节点是集群内外交流的一个Endpoint。

边缘节点要考虑两个问题

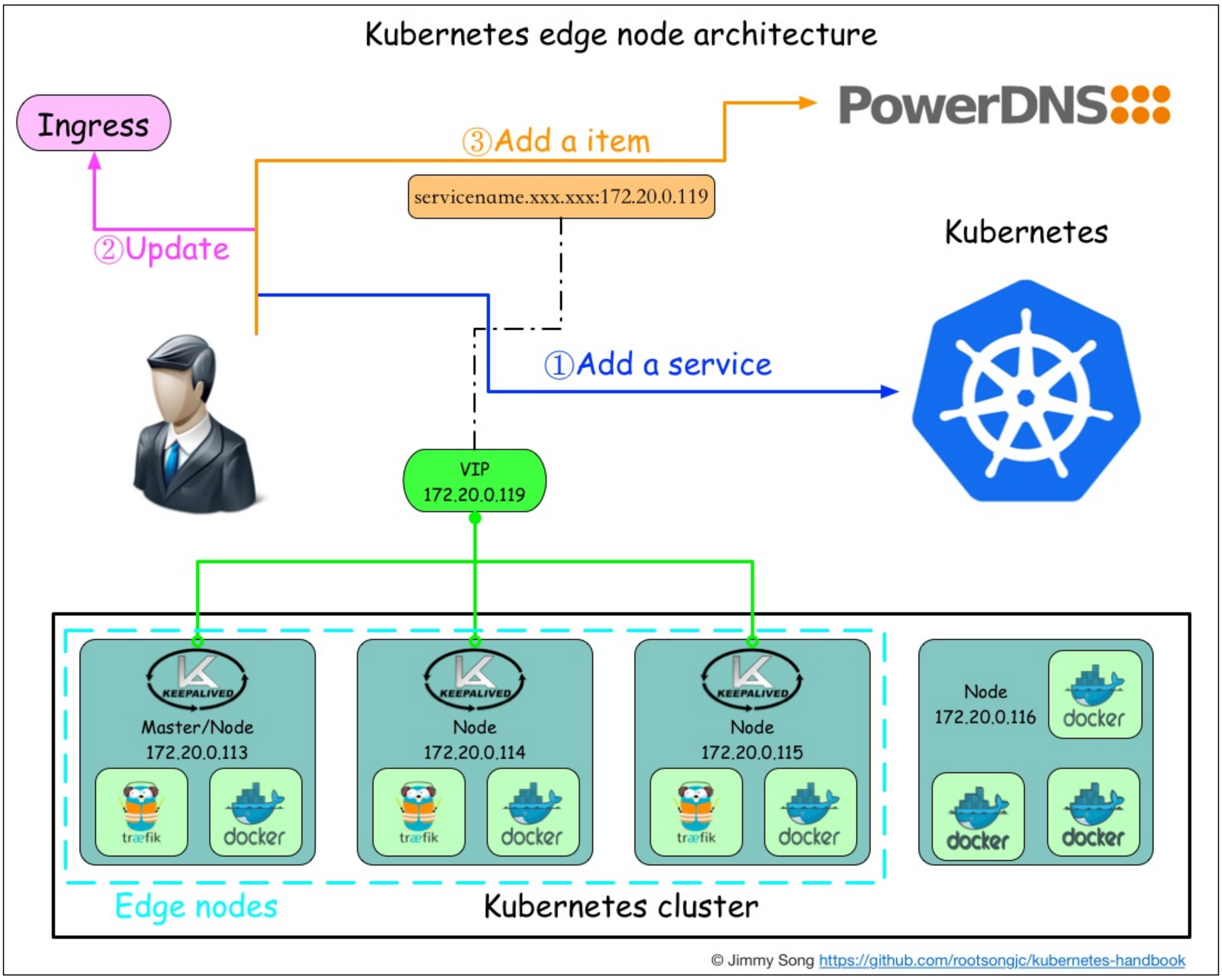
- 边缘节点的高可用，不能有单点故障，否则整个kubernetes集群将不可用
- 对外的一致暴露端口，即只能有一个外网访问IP和端口

架构

为了满足边缘节点的以上需求，我们使用keepalived来实现。

在Kubernetes中添加了service的同时，在DNS中增加一个记录，这条记录需要跟ingress中的 `host` 字段相同，IP地址即VIP的地址，本示例中是 `172.20.0.119` ，这样集群外部就可以通过service的DNS名称来访问服务了。

选择Kubernetes的三个node作为边缘节点，并安装keepalived，下图展示了边缘节点的配置，同时展示了向Kubernetes中添加服务的过程。



图片 - 边缘节点架构

准备

复用kubernetes测试集群的三台主机。

172.20.0.113

172.20.0.114

172.20.0.115

安装

使用keepalived管理VIP，VIP是使用IPVS创建的，IPVS已经成为linux内核的模块，不需要安装

LVS的工作原理请参考：http://www.cnblogs.com/codebean/archive/2011/07/25/2116043.html

不使用镜像方式安装了，直接手动安装，指定三个节点为边缘节点（Edge node）。

因为我们的测试集群一共只有三个node，所有在在三个node上都要安装keepalived和ipvsadmin。

```
yum install keepalived ipvsadm
```

配置说明

需要对原先的traefik ingress进行改造，从以Deployment方式启动改成DeamonSet。还需要指定一个与node在同一网段的IP地址作为VIP，我们指定成172.20.0.119，配置keepalived前需要先保证这个IP没有被分配。。

- Traefik以DaemonSet的方式启动
- 通过nodeSelector选择边缘节点
- 通过hostPort暴露端口
- 当前VIP漂移到了172.20.0.115上
- Traefik根据访问的host和path配置，将流量转发到相应的service上

配置keepalived

参考基于keepalived 实现VIP转移，lvs，nginx的高可用，配置keepalived。

keepalived的官方配置文档见：http://keepalived.org/pdf/UserGuide.pdf

配置文件 `/etc/keepalived/keepalived.conf` 文件内容如下：

```
! Configuration File for keepalived

global_defs {
   notification_email {
     root@localhost
   }
   notification_email_from kaadmin@localhost
   smtp_server 127.0.0.1
   smtp_connect_timeout 30
```

```
    router_id LVS_DEVEL
}

vrrp_instance VI_1 {
    state MASTER
    interface eth0
    virtual_router_id 51
    priority 100
    advert_int 1
    authentication {
        auth_type PASS
        auth_pass 1111
    }
    virtual_ipaddress {
        172.20.0.119
    }
}

virtual_server 172.20.0.119 80{
    delay_loop 6
    lb_algo loadbalance
    lb_kind DR
    nat_mask 255.255.255.0
    persistence_timeout 0
    protocol TCP

    real_server 172.20.0.113 80{
        weight 1
        TCP_CHECK {
        connect_timeout 3
        }
    }
    real_server 172.20.0.114 80{
        weight 1
        TCP_CHECK {
        connect_timeout 3
        }
    }
    real_server 172.20.0.115 80{
        weight 1
        TCP_CHECK {
        connect_timeout 3
        }
    }
}
```

`Realserver` 的IP和端口即traefik供外网访问的IP和端口。

将以上配置分别拷贝到另外两台node的 `/etc/keepalived` 目录下。

我们使用转发效率最高的 `lb_kind DR` 直接路由方式转发，使用TCP_CHECK来检测real_server的health。

设置keepalived为开机自启动：

```
chkconfig keepalived on
```

启动**keepalived**

```
systemctl start keepalived
```

三台node都启动了keepalived后，观察eth0的IP，会在三台node的某一台上发现一个VIP是172.20.0.119。

```
$ ip addr show eth0
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq sta
te UP qlen 1000
    link/ether f4:e9:d4:9f:6b:a0 brd ff:ff:ff:ff:ff:ff
    inet 172.20.0.115/17 brd 172.20.127.255 scope global eth0
       valid_lft forever preferred_lft forever
    inet 172.20.0.119/32 scope global eth0
       valid_lft forever preferred_lft forever
```

关掉拥有这个VIP主机上的keepalived，观察VIP是否漂移到了另外两台主机的其中之一上。

改造Traefik

在这之前我们启动的traefik使用的是deployment，只启动了一个pod，无法保证高可用（即需要将pod固定在某一台主机上，这样才能对外提供一个唯一的访问地址），现在使用了keepalived就可以通过VIP来访问traefik，同时启动多个traefik的pod保证高可用。

配置文件 `traefik.yaml` 内容如下：

```yaml
apiVersion: extensions/v1beta1
kind: DaemonSet
metadata:
  name: traefik-ingress-lb
  namespace: kube-system
  labels:
    k8s-app: traefik-ingress-lb
spec:
  template:
    metadata:
      labels:
        k8s-app: traefik-ingress-lb
        name: traefik-ingress-lb
    spec:
      terminationGracePeriodSeconds: 60
      hostNetwork: true
      restartPolicy: Always
      serviceAccountName: ingress
      containers:
      - image: traefik
        name: traefik-ingress-lb
        resources:
          limits:
            cpu: 200m
            memory: 30Mi
          requests:
            cpu: 100m
            memory: 20Mi
        ports:
        - name: http
          containerPort: 80
          hostPort: 80
        - name: admin
          containerPort: 8580
          hostPort: 8580
        args:
        - --web
        - --web.address=:8580
        - --kubernetes
      nodeSelector:
        edgenode: "true"
```

注意，我们使用了 `nodeSelector` 选择边缘节点来调度traefik-ingress-lb运行在它上面，所有你需要使用：

```
kubectl label nodes 172.20.0.113 edgenode=true
kubectl label nodes 172.20.0.114 edgenode=true
kubectl label nodes 172.20.0.115 edgenode=true
```

给三个node打标签。

查看DaemonSet的启动情况：

```
$ kubectl -n kube-system get ds
NAME                 DESIRED   CURRENT   READY     UP-TO-DATE
AVAILABLE   NODE-SELECTOR                              AGE
traefik-ingress-lb   3         3         3         3
3           edgenode=true                              2h
```

现在就可以在外网通过172.20.0.119:80来访问到traefik ingress了。

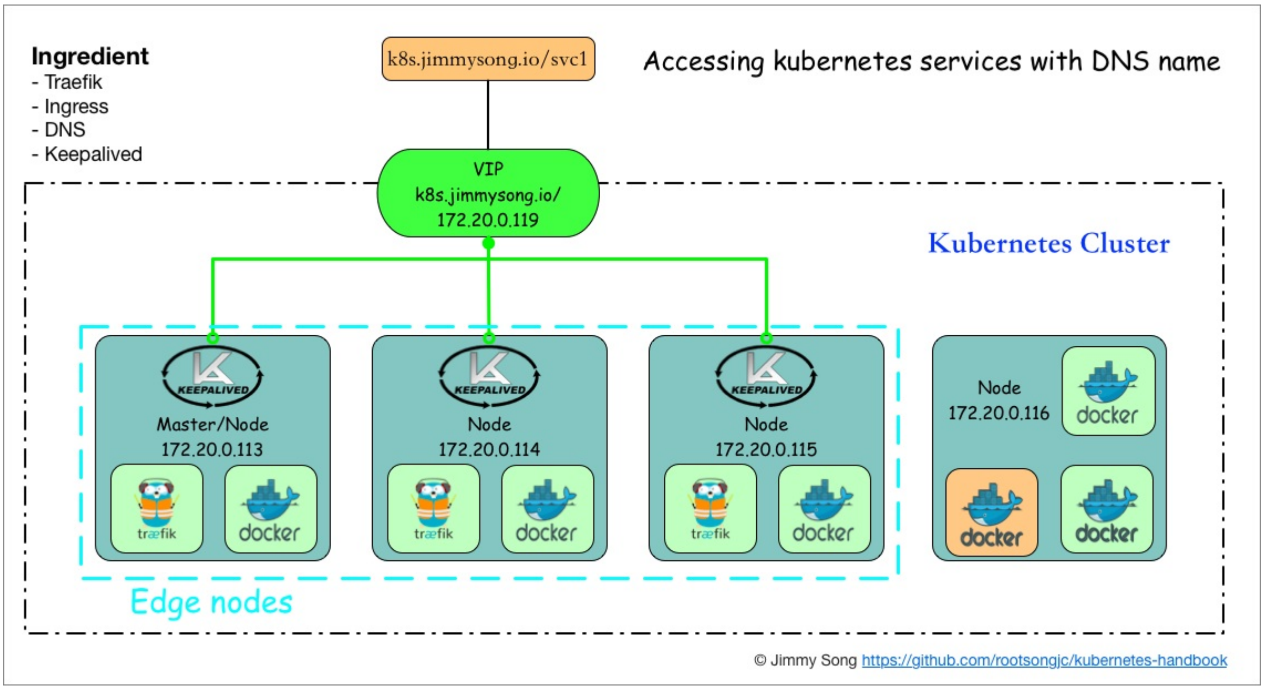
使用域名访问Kubernetes中的服务

现在我们已经部署了以下服务：

- 三个边缘节点，使用Traefik作为Ingress controller
- 使用keepalived做的VIP（虚拟IP）172.20.0.119

这样在访问该IP的时候通过指定不同的 `Host` 来路由到kubernetes后端服务。这种方式访问每个Service时都需要指定 `Host` ，而同一个项目中的服务一般会在同一个Ingress中配置，使用 `Path` 来区分Service已经足够，这时候只要为VIP（172.20.0.119）来配置一个域名，所有的外部访问直接通过该域名来访问即可。

如下图所示：



图片 - 使用域名来访问*Kubernetes*中的服务

参考

kube-keepalived-vip

http://www.keepalived.org/

keepalived工作原理与配置说明

LVS简介及使用

基于keepalived 实现VIP转移，lvs，nginx的高可用

安装Nginx ingress

Nginx ingress 使用ConfigMap来管理Nginx配置，nginx是大家熟知的代理和负载均衡软件，比起Traefik来说功能更加强大.

我们使用helm来部署，chart保存在私有的仓库中，请确保您已经安装和配置好helm，helm安装使用见使用Helm管理kubernetes应用。

镜像准备

安装时需要用到的镜像有：

- sophos/nginx-vts-exporter:v0.6
- gcr.io/google_containers/nginx-ingress-controller:0.9.0-beta.15
- gcr.io/google_containers/defaultbackend:1.3

gcr.io中的那个两个镜像我复制了一份到时速云，可供大家下载：

- index.tenxcloud.com/jimmy/defaultbackend:1.3
- index.tenxcloud.com/jimmy/nginx-ingress-controller:0.9.0-beta.15

Docker hub上的那个镜像可以直接下载，所有的安装时需要的配置保存在../manifests/nginx-ingress目录下。

步骤详解

安装**nginx-ingress chart**到本地**repo**中

修改 `values.yaml` 配置，启用RBAC支持，相关配置见nginx-ingress chart。

```
helm package .
```

查看**niginx-ingress**

```
$ helm search nginx-ingress
NAME                    VERSION     DESCRIPTION
local/nginx-ingress     0.8.9       An nginx Ingress controller t
hat uses ConfigMap...
stable/nginx-ingress    0.8.9       An nginx Ingress controller t
hat uses ConfigMap...
stable/nginx-lego       0.3.0       Chart for nginx-ingress-contr
oller and kube-lego
```

使用helm部署nginx-ingress

```
$ helm install --name nginx-ingress local/nginx-ingress
NAME:   nginx-ingress
LAST DEPLOYED: Fri Oct 27 18:26:58 2017
NAMESPACE: default
STATUS: DEPLOYED

RESOURCES:
==> rbac.authorization.k8s.io/v1beta1/Role
NAME                          KIND
nginx-ingress-nginx-ingress  Role.v1beta1.rbac.authorization.k8s
.io

==> rbac.authorization.k8s.io/v1beta1/RoleBinding
nginx-ingress-nginx-ingress  RoleBinding.v1beta1.rbac.authorizat
ion.k8s.io

==> v1/Service
NAME                                          CLUSTER-IP      EXT
ERNAL-IP  PORT(S)                      AGE
nginx-ingress-nginx-ingress-controller       10.254.100.108  <no
des>       80:30484/TCP,443:31053/TCP  1s
nginx-ingress-nginx-ingress-default-backend  10.254.58.156   <no
ne>        80/TCP                       1s

==> extensions/v1beta1/Deployment
NAME                                          DESIRED  CURRENT  U
P-TO-DATE  AVAILABLE  AGE
nginx-ingress-nginx-ingress-default-backend  1        1        1
           0          1s
nginx-ingress-nginx-ingress-controller       1        1        1
           0          1s

==> v1/ConfigMap
NAME                                    DATA  AGE
nginx-ingress-nginx-ingress-controller  1     1s

==> v1/ServiceAccount
NAME                          SECRETS  AGE
nginx-ingress-nginx-ingress  1        1s
```

```
==> rbac.authorization.k8s.io/v1beta1/ClusterRole
NAME                           KIND
nginx-ingress-nginx-ingress  ClusterRole.v1beta1.rbac.authorizat
ion.k8s.io

==> rbac.authorization.k8s.io/v1beta1/ClusterRoleBinding
nginx-ingress-nginx-ingress  ClusterRoleBinding.v1beta1.rbac.aut
horization.k8s.io


NOTES:
The nginx-ingress controller has been installed.
Get the application URL by running these commands:
  export HTTP_NODE_PORT=$(kubectl --namespace default get servic
es -o jsonpath="{.spec.ports[0].nodePort}" nginx-ingress-nginx-i
ngress-controller)
  export HTTPS_NODE_PORT=$(kubectl --namespace default get servi
ces -o jsonpath="{.spec.ports[1].nodePort}" nginx-ingress-nginx-
ingress-controller)
  export NODE_IP=$(kubectl --namespace default get nodes -o json
path="{.items[0].status.addresses[1].address}")

  echo "Visit http://$NODE_IP:$HTTP_NODE_PORT to access your app
lication via HTTP."
  echo "Visit https://$NODE_IP:$HTTPS_NODE_PORT to access your a
pplication via HTTPS."

An example Ingress that makes use of the controller:

  apiVersion: extensions/v1beta1
  kind: Ingress
  metadata:
    annotations:
      kubernetes.io/ingress.class: nginx
    name: example
    namespace: foo
  spec:
    rules:
      - host: www.example.com
        http:
          paths:
            - backend:
                serviceName: exampleService
                servicePort: 80
              path: /
    # This section is only required if TLS is to be enabled for
the Ingress
    tls:
        - hosts:
            - www.example.com
          secretName: example-tls
```

```
If TLS is enabled for the Ingress, a Secret containing the certi
ficate and key must also be provided:

  apiVersion: v1
  kind: Secret
  metadata:
    name: example-tls
    namespace: foo
  data:
    tls.crt: <base64 encoded cert>
    tls.key: <base64 encoded key>
  type: kubernetes.io/tls
```

访问Nginx

首先获取Nginx的地址，从我们使用helm安装nginx-ingress命令的输出中那个可以看到提示，根据提示执行可以看到nginx的http和https地址：

```
  export HTTP_NODE_PORT=$(kubectl --namespace default get servic
es -o jsonpath="{.spec.ports[0].nodePort}" nginx-ingress-nginx-i
ngress-controller)
  export HTTPS_NODE_PORT=$(kubectl --namespace default get servi
ces -o jsonpath="{.spec.ports[1].nodePort}" nginx-ingress-nginx-
ingress-controller)
  export NODE_IP=$(kubectl --namespace default get nodes -o json
path="{.items[0].status.addresses[1].address}")

  echo "Visit http://$NODE_IP:$HTTP_NODE_PORT to access your app
lication via HTTP."
  echo "Visit https://$NODE_IP:$HTTPS_NODE_PORT to access your a
pplication via HTTPS."
  Visit http://172.20.0.113:30484 to access your application via
 HTTP.
  Visit https://172.20.0.113:31053 to access your application vi
a HTTPS.
```

- http地址：http://172.20.0.113:30484
- https地址：https://172.20.0.113:31053

我们分别在http和https地址上测试一下：

- `/healthz` 返回200
- `/` 返回404错误

```
curl -v http://172.20.0.113:30484/healthz
# 返回200
curl -v http://172.20.0.113:30484/
# 返回404
curl -v --insecure http://172.20.0.113:30484/healthz
# 返回200
curl -v --insecure http://172.20.0.113:30484/
# 返回404
```

删除nginx-ingress

```
helm delete --purge nginx-ingress
```

使用 `--purge` 参数可以彻底删除release不留下记录，否则下一次部署的时候不能使用重名的release。

参考

- Ingress-nginx github
- Nginx chart configuration
- 使用Helm管理kubernetes应用

安装配置 DNS

DNS 组件作为 Kubernetes 中服务注册和发现的一个必要组件，起着举足轻重的作用，是我们在安装好 Kubernetes 集群后部署的第一个容器化应用。

安装配置kube-dns

在我们安装Kubernetes集群的时候就已经安装了kube-dns插件，这个插件也是官方推荐安装的。通过将 Service 注册到 DNS 中，Kuberentes 可以为我们提供一种简单的服务注册发现与负载均衡方式。

CoreDNS作为CNCF中的托管的一个项目，在Kuberentes1.9版本中，使用kubeadm方式安装的集群可以通过以下命令直接安装CoreDNS。

```
kubeadm init --feature-gates=CoreDNS=true
```

您也可以使用CoreDNS替换Kubernetes插件kube-dns，可以使用 Pod 部署也可以独立部署，请参考Using CoreDNS for Service Discovery，下文将介绍如何配置kube-dns。

kube-dns

kube-dns是Kubernetes中的一个内置插件，目前作为一个独立的开源项目维护，见https://github.com/kubernetes/dns。

下文中给出了配置 DNS Pod 的提示和定义 DNS 解析过程以及诊断 DNS 问题的指南。

前提要求

- Kubernetes 1.6 及以上版本。
- 集群必须使用 `kube-dns` 插件进行配置。

kube-dns 介绍

从 Kubernetes v1.3 版本开始，使用 cluster add-on 插件管理器回自动启动内置的 DNS。

Kubernetes DNS pod 中包括 3 个容器：

- `kubedns` ： `kubedns` 进程监视 Kubernetes master 中的 Service 和 Endpoint 的变化，并维护内存查找结构来服务DNS请求。
- `dnsmasq` ： `dnsmasq` 容器添加 DNS 缓存以提高性能。
- `sidecar` ： `sidecar` 容器在执行双重健康检查（针对 `dnsmasq` 和 `kubedns` ）时提供单个健康检查端点（监听在10054端口）。

DNS pod 具有静态 IP 并作为 Kubernetes 服务暴露出来。该静态 IP 分配后，kubelet 会将使用 `--cluster-dns = <dns-service-ip>` 标志配置的 DNS 传递给每个容器。

DNS 名称也需要域名。本地域可以使用标志 `--cluster-domain = <default-local-domain>` 在 kubelet 中配置。

Kubernetes集群DNS服务器基于 SkyDNS 库。它支持正向查找（A 记录），服务查找（SRV 记录）和反向 IP 地址查找（PTR 记录）

kube-dns 支持的 DNS 格式

kube-dns 将分别为 service 和 pod 生成不同格式的 DNS 记录。

Service

- A记录：生成 `my-svc.my-namespace.svc.cluster.local` 域名，解析成 IP 地址，分为两种情况：
  - 普通 Service：解析成 ClusterIP
  - Headless Service：解析为指定 Pod 的 IP 列表
- SRV记录：为命名的端口（普通 Service 或 Headless Service）生成 `_my-port-name._my-port-protocol.my-svc.my-namespace.svc.cluster.local` 的域名

Pod

- A记录：生成域名 `pod-ip.my-namespace.pod.cluster.local`

kube-dns 存根域名

可以在 Pod 中指定 hostname 和 subdomain： `hostname.custom-subdomain.default.svc.cluster.local` ，例如：

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: busybox
  labels:
    name: busybox
spec:
  hostname: busybox-1
  subdomain: busybox-subdomain
  containers:
  name: busybox
  - image: busybox
    command:
    - sleep
    - "3600"
```

该 Pod 的域名是 `busybox-1.busybox-subdomain.default.svc.cluster.local` 。

继承节点的 DNS

运行 Pod 时，kubelet 将预先配置集群 DNS 服务器到 Pod 中，并搜索节点自己的 DNS 设置路径。如果节点能够解析特定于较大环境的 DNS 名称，那么 Pod 应该也能够解析。请参阅下面的已知问题以了解警告。

如果您不想要这个，或者您想要为 Pod 设置不同的 DNS 配置，您可以给 kubelet 指定 `--resolv-conf` 标志。将该值设置为 "" 意味着 Pod 不继承 DNS。将其设置为有效的文件路径意味着 kubelet 将使用此文件而不是 `/etc/resolv.conf` 用于 DNS 继承。

配置存根域和上游 DNS 服务器

通过为 kube-dns （ `kube-system:kube-dns` ）提供一个 ConfigMap，集群管理员能够指定自定义存根域和上游 nameserver。

例如，下面的 ConfigMap 建立了一个 DNS 配置，它具有一个单独的存根域和两个上游 nameserver：

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: kube-dns
  namespace: kube-system
data:
  stubDomains: |
    {“acme.local”: [“1.2.3.4”]}
  upstreamNameservers: |
    [“8.8.8.8”, “8.8.4.4”]
```

如上面指定的那样，带有“.acme.local”后缀的 DNS 请求被转发到 1.2.3.4 处监听的 DNS。Google Public DNS 为上游查询提供服务。

下表描述了如何将具有特定域名的查询映射到其目标DNS服务器：

域名	响应查询的服务器
kubernetes.default.svc.cluster.local	kube-dns
foo.acme.local	自定义 DNS (1.2.3.4)
widget.com	上游 DNS (8.8.8.8 或 8.8.4.4)

查看 ConfigMap 选项 获取更多关于配置选项格式的详细信息。

对 Pod 的影响

自定义的上游名称服务器和存根域不会影响那些将自己的 `dnsPolicy` 设置为 `Default` 或者 `None` 的 Pod。

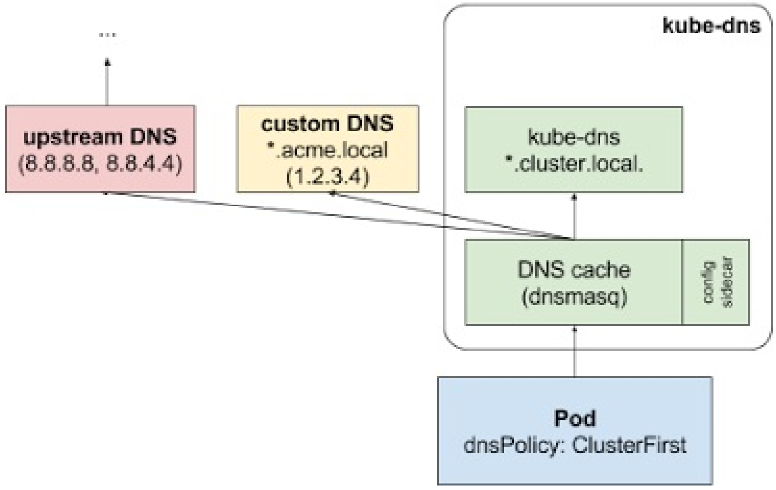
如果 Pod 的 `dnsPolicy` 设置为“ `ClusterFirst` ”，则其名称解析将按其他方式处理，具体取决于存根域和上游 DNS 服务器的配置。

未进行自定义配置：没有匹配上配置的集群域名后缀的任何请求，例如“www.kubernetes.io”，将会被转发到继承自节点的上游 nameserver。

进行自定义配置：如果配置了存根域和上游 DNS 服务器（和在 前面例子 配置的一样），DNS 查询将根据下面的流程进行路由：

1. 查询首先被发送到 kube-dns 中的 DNS 缓存层。
2. 从缓存层，检查请求的后缀，并转发到合适的 DNS 上，基于如下的示例：
    - 具有集群后缀的名字（例如“.cluster.local”）：请求被发送到 kube-dns。

- 具有存根域后缀的名字（例如 “.acme.local”）：请求被发送到配置的自定义 DNS 解析器（例如：监听在 1.2.3.4）。
- 不具有能匹配上后缀的名字（例如 “widget.com”）：请求被转发到上游 DNS（例如：Google 公共 DNS 服务器，8.8.8.8 和 8.8.4.4）。



图片 - *DNS lookup flow*

ConfigMap 选项

kube-dns `kube-system:kube-dns` ConfigMap 的选项如下所示：

字段	格式	描述
`stubDomains` （可选）	使用 DNS 后缀 key 的 JSON map（例如 “acme.local”），以及 DNS IP 的 JSON 数组作为 value。	目标 nameserver 可能是一个 Kubernetes Service。例如，可以运行自己的 dnsmasq 副本，将 DNS 名字暴露到 ClusterDNS namespace 中。
`upstreamNameservers` （可选）	DNS IP 的 JSON 数组。	注意：如果指定，则指定的值会替换掉被默认从节点的 `/etc/resolv.conf` 中获取到的 nameserver。限制：最多可以指定三个上游 nameserver。

示例

示例：存根域

在这个例子中，用户有一个 Consul DNS 服务发现系统，他们希望能够与 kube-dns 集成起来。 Consul 域名服务器地址为 10.150.0.1，所有的 Consul 名字具有后缀 `.consul.local` 。要配置 Kubernetes，集群管理员只需要简单地创建一个 ConfigMap 对象，如下所示：

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: kube-dns
  namespace: kube-system
data:
  stubDomains: |
    {"consul.local": ["10.150.0.1"]}
```

注意，集群管理员不希望覆盖节点的上游 nameserver，所以他们不会指定可选的 `upstreamNameservers` 字段。

示例：上游 **nameserver**

在这个示例中，集群管理员不希望显式地强制所有非集群 DNS 查询进入到他们自己的 nameserver 172.16.0.1。 而且这很容易实现：他们只需要创建一个 ConfigMap，`upstreamNameservers` 字段指定期望的 nameserver 即可。

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: kube-dns
  namespace: kube-system
data:
  upstreamNameservers: |
    ["172.16.0.1"]
```

调试 **DNS** 解析

创建一个简单的 **Pod** 用作测试环境

创建一个名为 busybox.yaml 的文件，其中包括以下内容：

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: busybox
  namespace: default
spec:
  containers:
  - name: busybox
    image: busybox
    command:
      - sleep
      - "3600"
    imagePullPolicy: IfNotPresent
  restartPolicy: Always
```

使用该文件创建 Pod 并验证其状态：

$ kubectl create -f busybox.yaml pod "busybox" created

$ kubectl get pods busybox NAME READY STATUS RESTARTS AGE busybox 1/1 Running 0

```
该 Pod 运行后，您可以在它的环境中执行 `nslookup`。如果您看到类似如下的输出
，表示 DNS 正在正确工作。

```bash
$ kubectl exec -ti busybox -- nslookup kubernetes.default
Server:    10.0.0.10
Address 1: 10.0.0.10

Name:      kubernetes.default
Address 1: 10.0.0.1
```

如果 `nslookup` 命令失败，检查如下内容：

首先检查本地 DNS 配置

查看下 resolv.conf 文件。（参考集成节点的 DNS和 下面的已知问题获取更多信息）

```
$ kubectl exec busybox cat /etc/resolv.conf
```
```

验证搜索路径和名称服务器设置如下（请注意，搜索路径可能因不同的云提供商而异）：

```
search default.svc.cluster.local svc.cluster.local cluster.local
 google.internal c.gce_project_id.internal
nameserver 10.0.0.10
options ndots:5
```

如果看到如下错误表明错误来自 kube-dns 或相关服务：

```
$ kubectl exec -ti busybox -- nslookup kubernetes.default
Server:    10.0.0.10
Address 1: 10.0.0.10

nslookup: can't resolve 'kubernetes.default'
```

或者

```
$ kubectl exec -ti busybox -- nslookup kubernetes.default
Server:    10.0.0.10
Address 1: 10.0.0.10 kube-dns.kube-system.svc.cluster.local

nslookup: can't resolve 'kubernetes.default'
```

检查 DNS pod 是否在运行

使用 `kubectl get pods` 命令验证 DNS pod 是否正在运行。

```
$ kubectl get pods --namespace=kube-system -l k8s-app=kube-dns
NAME                    READY     STATUS    RESTARTS   AGE
...
kube-dns-v19-ezo1y      3/3       Running   0           1h
...
```

如果您看到没有 Pod 运行或者 Pod 处于 失败/完成 状态，DNS 插件可能没有部署到您的当前环境中，您需要手动部署。

检查 DNS pod 中的错误

使用 `kubectl logs` 命令查看 DNS 守护进程的日志。

```
$ kubectl logs --namespace=kube-system $(kubectl get pods --name
space=kube-system -l k8s-app=kube-dns -o name) -c kubedns
$ kubectl logs --namespace=kube-system $(kubectl get pods --name
space=kube-system -l k8s-app=kube-dns -o name) -c dnsmasq
$ kubectl logs --namespace=kube-system $(kubectl get pods --name
space=kube-system -l k8s-app=kube-dns -o name) -c sidecar
```

看看有没有可疑的日志。以字母“ `W` ”，“ `E` ”，“ `F` ”开头的代表警告、错误和失败。请搜索具有这些日志级别的条目，并使用 kubernetes issues来报告意外错误。

DNS 服务启动了吗？

使用 `kubectl get service` 命令验证 DNS 服务是否启动。

```
$ kubectl get svc --namespace=kube-system
NAME          CLUSTER-IP     EXTERNAL-IP   PORT(S)             A
GE
...
kube-dns      10.0.0.10      <none>        53/UDP,53/TCP
1h
...
```

如果您已经创建了该服务或它本应该默认创建但没有出现，参考调试服务获取更多信息。

DNS 端点暴露出来了吗？

您可以使用 `kubectl get endpoints` 命令验证 DNS 端点是否被暴露。

```
$ kubectl get ep kube-dns --namespace=kube-system
NAME       ENDPOINTS                       AGE
kube-dns   10.180.3.17:53,10.180.3.17:53    1h
```

如果您没有看到端点，查看调试服务文档中的端点部分。

获取更多的 Kubernetes DNS 示例，请参考 Kubernetes GitHub 仓库中的cluster-dns示例。

已知问题

Kubernetes安装时不会将节点的 resolv.conf 文件配置为默认使用集群 DNS，因为该过程本身是特定于发行版的。这一步应该放到最后实现。

Linux 的 libc 不可思议的卡住（查看该2005年起暴出来的bug）限制只能有 3 个 DNS `nameserver` 记录和 6 个 DNS `search` 记录。Kubernetes 需要消耗 1 个 `nameserver` 记录和 3 个 `search` 记录。这意味着如果本地安装已经使用 3 个 `nameserver` 或使用 3 个以上的 `search` 记录，那么其中一些设置将会丢失。有个部分解决该问题的方法，就是节点可以运行 `dnsmasq` ，它将提供更多的 `nameserver` 条目，但不会有更多的 `search` 条目。您也可以使用 kubelet 的 `--resolv-conf` 标志。

如果您使用的是 Alpine 3.3 或更低版本作为基础映像，由于已知的 Alpine 问题，DNS 可能无法正常工作。点击这里查看更多信息。

Kubernetes 集群联邦（多可用区支持）

Kubernetes 1.3 版本起引入了支持多站点 Kubernetes 安装的集群联邦支持。这需要对 Kubernetes 集群 DNS 服务器处理 DNS 查询的方式进行一些小的（向后兼容的）更改，以便于查找联邦服务（跨多个 Kubernetes 集群）。有关集群联邦和多站点支持的更多详细信息，请参阅集群联邦管理员指南。

参考

- Configure DNS Service
- Service 和 Pod 的 DNS
- 自动扩容集群中的 DNS 服务
- Using CoreDNS for Service Discovery

安装配置CoreDNS

CoreDNS可以在具有标准的Kube-DNS的Kubernetes集群中运行。作为Kubernetes的插件使用，CoreDNS将从 Kubernetes集群中读取区（zone）数据。它实现了为Kubernetes的DNS服务发现定义的规范：Kubernetes DNS-Based Service Discovery。

部署CoreDNS

部署 CoreDNS 需要使用到官方提供的两个文件 deploy.sh和coredns.yaml.sed（这两个文件已经放入manifest的coredns中）

`deploy.sh` 是一个用于在已经运行kube-dns的集群中生成运行CoreDNS部署文件（manifest）的工具脚本。它使用 `coredns.yaml.sed` 文件作为模板，创建一个ConfigMap和CoreDNS的deployment，然后更新集群中已有的kube-dns 服务的selector使用CoreDNS的deployment。重用已有的服务并不会在服务的请求中发生冲突。

`deploy.sh` 文件并不会删除kube-dns的deployment或者replication controller。如果要删除kube-dns，你必须在部署CoreDNS后手动的删除kube-dns。

你需要仔细测试manifest文件，以确保它能够对你的集群正常运行。这依赖于你的怎样构建你的集群以及你正在运行的集群版本。 对manifest文件做一些修改是有比要的。

在最佳的案例场景中，使用CoreDNS替换Kube-DNS只需要使用下面的两个命令：

```
$ ./deploy.sh | kubectl apply -f -
$ kubectl delete --namespace=kube-system deployment kube-dns
```

注意：我们建议在部署CoreDNS后删除kube-dns。否则如果CoreDNS和kube-dns同时运行，服务查询可能会随机的在CoreDNS和kube-dns之间产生。

对于非RBAC部署，你需要编辑生成的结果yaml文件：

1. 从yaml文件的 `Deployment` 部分删除 `serviceAccountName: coredns`
2. 删除 `ServiceAccount` 、 `ClusterRole` 和 `ClusterRoleBinding` 部分

参考

- Kubernetes DNS-Based Service Discovery

Updated at 2018-05-09 14:15:47

运维管理

将集群部署到生产环境后就不得不考虑运维管理问题。运维管理问题主要包括如下几个方面：

- 监控：包括 kubernetes 本身组件和 Pod、应用的监控
- 日志收集：包括 kubernetes 本身组件的日志，应用的日志
- 审计：用户对集群操作的审计
- 安全：用户权限的管理和镜像漏洞扫描

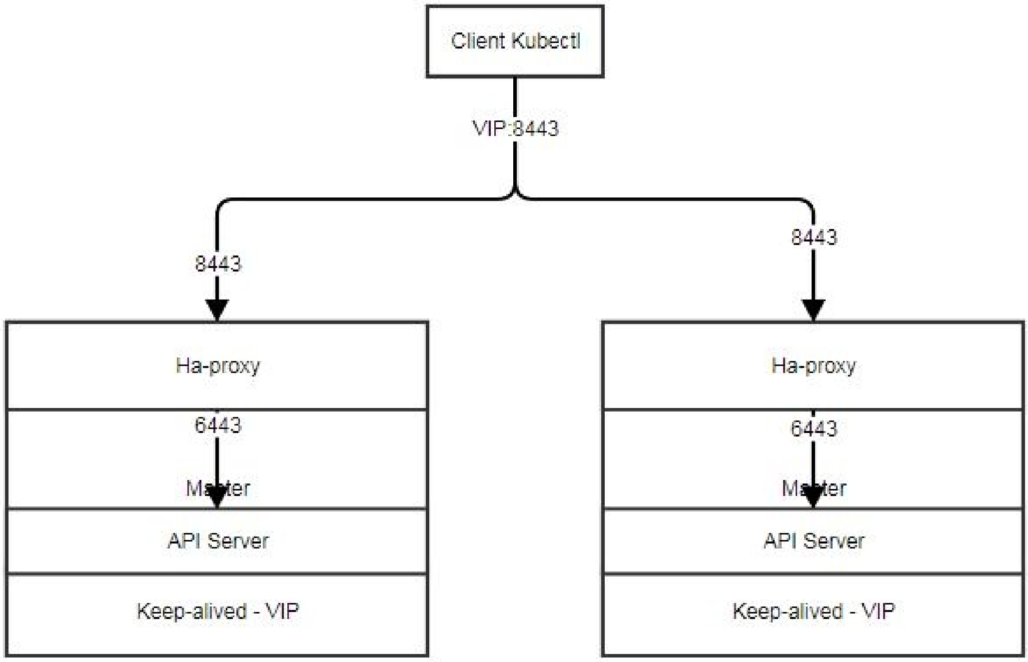
Master节点高可用

作者：mendickxiao

经过部署Kubernetes集群章节我们已经可以顺利的部署一个集群用于开发和测试，但是要应用到生产就就不得不考虑master节点的高可用问题，因为现在我们的master节点上的几个服务 `kube-apiserver` 、 `kube-scheduler` 和 `kube-controller-manager` 都是单点的而且都位于同一个节点上，一旦master节点宕机，虽然不应答当前正在运行的应用，将导致kubernetes集群无法变更。本文将引导你创建一个高可用的master节点。

在大神gzmzj的ansible创建kubernetes集群神作中有讲到如何配置多个Master，但是在实践过程中还是遇到不少坑。需要将坑填上才能工作。 神作链接地址：集群规划和基础参数设定。

按照神作的描述，实际上是通过keepalived + haproxy实现的，其中keepalived是提供一个VIP，通过VIP关联所有的Master节点；然后haproxy提供端口转发功能。由于VIP还是存在Master的机器上的，默认配置API Server的端口是6443，所以我们需要将另外一个端口关联到这个VIP上，一般用8443。



图片 - *Master HA*架构图

根据神作的实践，我发现需要在Master手工安装keepalived, haproxy。

```
yum install keepalived
yum install haproxy
```

需要将HAProxy默认的配置文件balance从source修改为 `roundrobin` 方式。haproxy的配置文件 `haproxy.cfg` 默认路径是 `/etc/haproxy/haproxy.cfg` 。另外需要手工创建 `/run/haproxy` 的目录，否则haproxy会启动失败。

注意

- bind绑定的就是VIP对外的端口号，这里是8443。
- balance指定的负载均衡方式是 `roundrobin` 方式，默认是source方式。在我的测试中，source方式不工作。
- server指定的就是实际的Master节点地址以及真正工作的端口号，这里是6443。有多少台Master就写多少条记录。

```
# haproxy.cfg sample
global
        log /dev/log    local0
        log /dev/log    local1 notice
        chroot /var/lib/haproxy
        *stats socket /run/haproxy/admin.sock mode 660 level adm
in
        stats timeout 30s
        user haproxy
        group haproxy
        daemon
        nbproc 1

defaults
        log     global
        timeout connect 5000
        timeout client  50000
        timeout server  50000

listen kube-master
        **bind 0.0.0.0:8443**
        mode tcp
        option tcplog
        **balance roundrobin**
        server s1 **Master 1的IP地址**:6443  check inter 10000 fa
ll 2 rise 2 weight 1
        server s2 **Master 2的IP地址**:6443  check inter 10000 fa
ll 2 rise 2 weight 1
```

修改keepalived的配置文件，配置正确的VIP。keepalived的配置文件 `keepalived.conf` 的默认路径是 `/etc/keepalived/keepalived.conf`

注意

- priority决定哪个Master是主，哪个Master是次。数字小的是主，数字大的是次。数字越小优先级越高。
- `virtual_router_id` 决定当前VIP的路由号，实际上VIP提供了一个虚拟的路由功能，该VIP在同一个子网内必须是唯一。
- virtual_ipaddress提供的就是VIP的地址，该地址在子网内必须是空闲未必分配的。

```
# keepalived.cfg sample

global_defs {
    router_id lb-backup
}

vrrp_instance VI-kube-master {
    state BACKUP
    **priority 110**
    dont_track_primary
    interface eth0
    **virtual_router_id 51**
    advert_int 3
    virtual_ipaddress {
        **10.86.13.36**
    }
}
```

配置好后，那么先启动主Master的keepalived和haproxy。

```
systemctl enable keepalived
systemctl start keepalived
systemctl enable haproxy
systemctl start haproxy
```

然后使用ip a s命令查看是否有VIP地址分配。如果看到VIP地址已经成功分配在eth0网卡上，说明keepalived启动成功。

```
[root@kube32 ~]# ip a s
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKN
OWN qlen 1
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq sta
te UP qlen 1000
    link/ether 00:50:56:a9:d5:be brd ff:ff:ff:ff:ff:ff
    inet 10.86.13.32/23 brd 10.86.13.255 scope global eth0
       valid_lft forever preferred_lft forever
    **inet 10.86.13.36/32 scope global eth0**
       valid_lft forever preferred_lft forever
    inet6 fe80::250:56ff:fea9:d5be/64 scope link
       valid_lft forever preferred_lft forever
```

更保险方法还可以通过 `systemctl status keepalived -l` 看看keepalived的状态

```
[root@kube32 ~]# systemctl status keepalived -l
● keepalived.service - LVS and VRRP High Availability Monitor
   Loaded: loaded (/usr/lib/systemd/system/keepalived.service; e
nabled; vendor preset: disabled)
   Active: active (running) since Thu 2018-02-01 10:24:51 CST; 1
 months 16 days ago
 Main PID: 13448 (keepalived)
   Memory: 6.0M
   CGroup: /system.slice/keepalived.service
           ├─13448 /usr/sbin/keepalived -D
           ├─13449 /usr/sbin/keepalived -D
           └─13450 /usr/sbin/keepalived -D

Mar 20 04:51:15 kube32 Keepalived_vrrp[13450]: VRRP_Instance(VI-
kube-master) Dropping received VRRP packet...
**Mar 20 04:51:18 kube32 Keepalived_vrrp[13450]: (VI-kube-master)
: ip address associated with VRID 51 not present in MASTER adver
t : 10.86.13.36
Mar 20 04:51:18 kube32 Keepalived_vrrp[13450]: bogus VRRP packet
 received on eth0 !!!**
```

然后通过systemctl status haproxy -l看haproxy的状态

```
[root@kube32 ~]# systemctl status haproxy -l
● haproxy.service - HAProxy Load Balancer
   Loaded: loaded (/usr/lib/systemd/system/haproxy.service; enab
led; vendor preset: disabled)
   Active: active (running) since Thu 2018-02-01 10:33:22 CST; 1
 months 16 days ago
 Main PID: 15116 (haproxy-systemd)
   Memory: 3.2M
   CGroup: /system.slice/haproxy.service
           ├─15116 /usr/sbin/haproxy-systemd-wrapper -f /etc/hap
roxy/haproxy.cfg -p /run/haproxy.pid
           ├─15117 /usr/sbin/haproxy -f /etc/haproxy/haproxy.cfg
 -p /run/haproxy.pid -Ds
           └─15118 /usr/sbin/haproxy -f /etc/haproxy/haproxy.cfg
 -p /run/haproxy.pid -Ds
```

这个时候通过kubectl version命令，可以获取到kubectl的服务器信息。

```
[root@kube32 ~]# kubectl version
**Client Version: version.Info{Major:"1", Minor:"9", GitVersion:
"v1.9.1", GitCommit:"3a1c9449a956b6026f075fa3134ff92f7d55f812",
GitTreeState:"clean", BuildDate:"2018-01-03T22:31:01Z", GoVersio
n:"go1.9.2", Compiler:"gc", Platform:"linux/amd64"}
Server Version: version.Info{Major:"1", Minor:"9", GitVersion:"v
1.9.1", GitCommit:"3a1c9449a956b6026f075fa3134ff92f7d55f812", Gi
tTreeState:"clean", BuildDate:"2018-01-03T22:18:41Z", GoVersion:
"go1.9.2", Compiler:"gc", Platform:"linux/amd64"}**
```

这个时候说明你的keepalived和haproxy都是成功的。这个时候你可以依次将你其他Master节点的keepalived和haproxy启动。 此时，你通过ip a s命令去查看其中一台Master（非主*Master*）的时候，你看不到VIP，这个是正常的，因为VIP永远只在主Master节点上，只有当主Master节点挂掉后，才会切换到其他Master节点上。

```
[root@kube31 ~]# ip a s
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKN
OWN qlen 1
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq sta
te UP qlen 1000
    link/ether 00:50:56:a9:07:23 brd ff:ff:ff:ff:ff:ff
    inet 10.86.13.31/23 brd 10.86.13.255 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::250:56ff:fea9:723/64 scope link
       valid_lft forever preferred_lft forever
```

在我的实践过程中，通过大神的脚本快速启动多个Master节点，会导致主Master始终获取不了VIP，当时的报错非常奇怪。后来经过我的研究发现，主Master获取VIP是需要时间的，如果多个Master同时启动，会导致冲突。这个不知道是否算是Keepalived的Bug。但是最稳妥的方式还是先启动一台主Master，等VIP确定后再启动其他Master比较靠谱。

Kubernetes通过Keepalived + Haproxy实现多个Master的高可用部署，你实际上可以采用其他方式，如外部的负载均衡方式。实际上Kubernetes的多个Master是没有主从的，都可以提供一致性服务。Keepalived + Haproxy实际上就是提供了一个简单的负载均衡方式。

GitbookUpdated at 2018-05-09 14:15:47

服务滚动升级

当有镜像发布新版本，新版本服务上线时如何实现服务的滚动和平滑升级？

如果你使用**ReplicationController**创建的pod可以使用 `kubectl rollingupdate` 命令滚动升级，如果使用的是**Deployment**创建的Pod可以直接修改yaml文件后执行 `kubectl apply` 即可。

Deployment已经内置了RollingUpdate strategy，因此不用再调用 `kubectl rollingupdate` 命令，升级的过程是先创建新版的pod将流量导入到新pod上后销毁原来的旧的pod。

Rolling Update适用于 `Deployment` 、 `Replication Controller` ，官方推荐使用Deployment而不再使用Replication Controller。

使用ReplicationController时的滚动升级请参考官网说明：https://kubernetes.io/docs/tasks/run-application/rolling-update-replication-controller/

ReplicationController与Deployment的关系

ReplicationController和Deployment的RollingUpdate命令有些不同，但是实现的机制是一样的，关于这两个kind的关系我引用了ReplicationController与Deployment的区别中的部分内容如下，详细区别请查看原文。

ReplicationController

Replication Controller为Kubernetes的一个核心内容，应用托管到Kubernetes之后，需要保证应用能够持续的运行，Replication Controller就是这个保证的key，主要的功能如下：

- 确保pod数量：它会确保Kubernetes中有指定数量的Pod在运行。如果少于指定数量的pod，Replication Controller会创建新的，反之则会删除掉多余的以保证Pod数量不变。
- 确保pod健康：当pod不健康，运行出错或者无法提供服务时，Replication Controller也会杀死不健康的pod，重新创建新的。

- 弹性伸缩 ：在业务高峰或者低峰期的时候，可以通过Replication Controller动态的调整pod的数量来提高资源的利用率。同时，配置相应的监控功能（Hroizontal Pod Autoscaler），会定时自动从监控平台获取Replication Controller关联pod的整体资源使用情况，做到自动伸缩。
- 滚动升级：滚动升级为一种平滑的升级方式，通过逐步替换的策略，保证整体系统的稳定，在初始化升级的时候就可以及时发现和解决问题，避免问题不断扩大。

Deployment

Deployment同样为Kubernetes的一个核心内容，主要职责同样是为了保证pod的数量和健康，90%的功能与Replication Controller完全一样，可以看做新一代的Replication Controller。但是，它又具备了Replication Controller之外的新特性：

- Replication Controller全部功能：Deployment继承了上面描述的Replication Controller全部功能。
- 事件和状态查看：可以查看Deployment的升级详细进度和状态。
- 回滚：当升级pod镜像或者相关参数的时候发现问题，可以使用回滚操作回滚到上一个稳定的版本或者指定的版本。
- 版本记录: 每一次对Deployment的操作，都能保存下来，给予后续可能的回滚使用。
- 暂停和启动：对于每一次升级，都能够随时暂停和启动。
- 多种升级方案：Recreate：删除所有已存在的pod,重新创建新的; RollingUpdate：滚动升级，逐步替换的策略，同时滚动升级时，支持更多的附加参数，例如设置最大不可用pod数量，最小升级间隔时间等等。

创建测试镜像

我们来创建一个特别简单的web服务，当你访问网页时，将输出一句版本信息。通过区分这句版本信息输出我们就可以断定升级是否完成。

所有配置和代码见../manifests/test/rolling-update-test目录。

Web服务的代码main.go

```
package main

import (
    "fmt"
    "log"
    "net/http"
)

func sayhello(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "This is version 1.") //这个写入到w的是输出到客户
端的
}

func main() {
    http.HandleFunc("/", sayhello) //设置访问的路由
    log.Println("This is version 1.")
    err := http.ListenAndServe(":9090", nil) //设置监听的端口
    if err != nil {
        log.Fatal("ListenAndServe: ", err)
    }
}
```

创建**Dockerfile**

```
FROM alpine:3.5
MAINTAINER Jimmy Song<rootsongjc@gmail.com>
ADD hellov2 /
ENTRYPOINT ["/hellov2"]
```

注意修改添加的文件的名称。

创建**Makefile**

修改镜像仓库的地址为你自己的私有镜像仓库地址。

修改 `Makefile` 中的 `TAG` 为新的版本号。

```
all: build push clean
.PHONY: build push clean

TAG = v1

# Build for linux amd64
build:
	GOOS=linux GOARCH=amd64 go build -o hello${TAG} main.go
	docker build -t harbor-001.jimmysong.io/library/hello:${TAG}
 .

# Push to tenxcloud
push:
	docker push harbor-001.jimmysong.io/library/hello:${TAG}

# Clean
clean:
	rm -f hello${TAG}
```

编译

```
make all
```

分别修改main.go中的输出语句、Dockerfile中的文件名称和Makefile中的TAG，创建两个版本的镜像。

测试

我们使用Deployment部署服务来测试。

配置文件 `rolling-update-test.yaml` ：

```yaml
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
    name: rolling-update-test
spec:
  replicas: 3
  template:
    metadata:
      labels:
        app: rolling-update-test
    spec:
      containers:
      - name: rolling-update-test
        image: harbor-001.jimmysong.io/library/hello:v1
        ports:
        - containerPort: 9090
---
apiVersion: v1
kind: Service
metadata:
  name: rolling-update-test
  labels:
    app: rolling-update-test
spec:
  ports:
  - port: 9090
    protocol: TCP
    name: http
  selector:
    app: rolling-update-test
```

部署service

```
kubectl create -f rolling-update-test.yaml
```

修改traefik ingress配置

在 `ingress.yaml` 文件中增加新service的配置。

```yaml
  - host: rolling-update-test.traefik.io
    http:
      paths:
      - path: /
        backend:
          serviceName: rolling-update-test
          servicePort: 9090
```

修改本地的host配置，增加一条配置：

```
172.20.0.119 rolling-update-test.traefik.io
```

注意：172.20.0.119是我们之前使用keepalived创建的VIP。

打开浏览器访问http://rolling-update-test.traefik.io将会看到以下输出：

```
This is version 1.
```

滚动升级

只需要将 `rolling-update-test.yaml` 文件中的 `image` 改成新版本的镜像名，然后执行：

```
kubectl apply -f rolling-update-test.yaml
```

也可以参考Kubernetes Deployment Concept中的方法，直接设置新的镜像。

```
kubectl set image deployment/rolling-update-test rolling-update-
test=harbor-001.jimmysong.io/library/hello:v2
```

或者使用 `kubectl edit deployment/rolling-update-test` 修改镜像名称后保存。

使用以下命令查看升级进度：

```
kubectl rollout status deployment/rolling-update-test
```

升级完成后在浏览器中刷新http://rolling-update-test.traefik.io将会看到以下输出：

```
This is version 2.
```

说明滚动升级成功。

使用ReplicationController创建的Pod如何RollingUpdate

以上讲解使用**Deployment**创建的Pod的RollingUpdate方式，那么如果使用传统的**ReplicationController**创建的Pod如何Update呢？

举个例子：

```
$ kubectl -n spark-cluster rolling-update zeppelin-controller --image harbor-001.jimmysong.io/library/zeppelin:0.7.1
Created zeppelin-controller-99be89dbbe5cd5b8d6feab8f57a04a8b
Scaling up zeppelin-controller-99be89dbbe5cd5b8d6feab8f57a04a8b from 0 to 1, scaling down zeppelin-controller from 1 to 0 (keep 1 pods available, don't exceed 2 pods)
Scaling zeppelin-controller-99be89dbbe5cd5b8d6feab8f57a04a8b up to 1
Scaling zeppelin-controller down to 0
Update succeeded. Deleting old controller: zeppelin-controller
Renaming zeppelin-controller-99be89dbbe5cd5b8d6feab8f57a04a8b to zeppelin-controller
replicationcontroller "zeppelin-controller" rolling updated
```

只需要指定新的镜像即可，当然你可以配置RollingUpdate的策略。

参考

- Rolling update机制解析
- Running a Stateless Application Using a Deployment
- Simple Rolling Update
- 使用kubernetes的deployment进行RollingUpdate

应用日志收集

前言

在进行日志收集的过程中，我们首先想到的是使用Logstash，因为它是ELK stack中的重要成员，但是在测试过程中发现，Logstash是基于JDK的，在没有产生日志的情况单纯启动Logstash就大概要消耗**500M**内存，在每个Pod中都启动一个日志收集组件的情况下，使用logstash有点浪费系统资源，经人推荐我们选择使用**Filebeat**替代，经测试单独启动Filebeat容器大约会消耗**12M**内存，比起logstash相当轻量级。

方案选择

Kubernetes官方提供了EFK的日志收集解决方案，但是这种方案并不适合所有的业务场景，它本身就有一些局限性，例如：

- 所有日志都必须是out前台输出，真实业务场景中无法保证所有日志都在前台输出
- 只能有一个日志输出文件，而真实业务场景中往往有多个日志输出文件
- Fluentd并不是常用的日志收集工具，我们更习惯用logstash，现使用filebeat替代
- 我们已经有自己的ELK集群且有专人维护，没有必要再在kubernetes上做一个日志收集服务

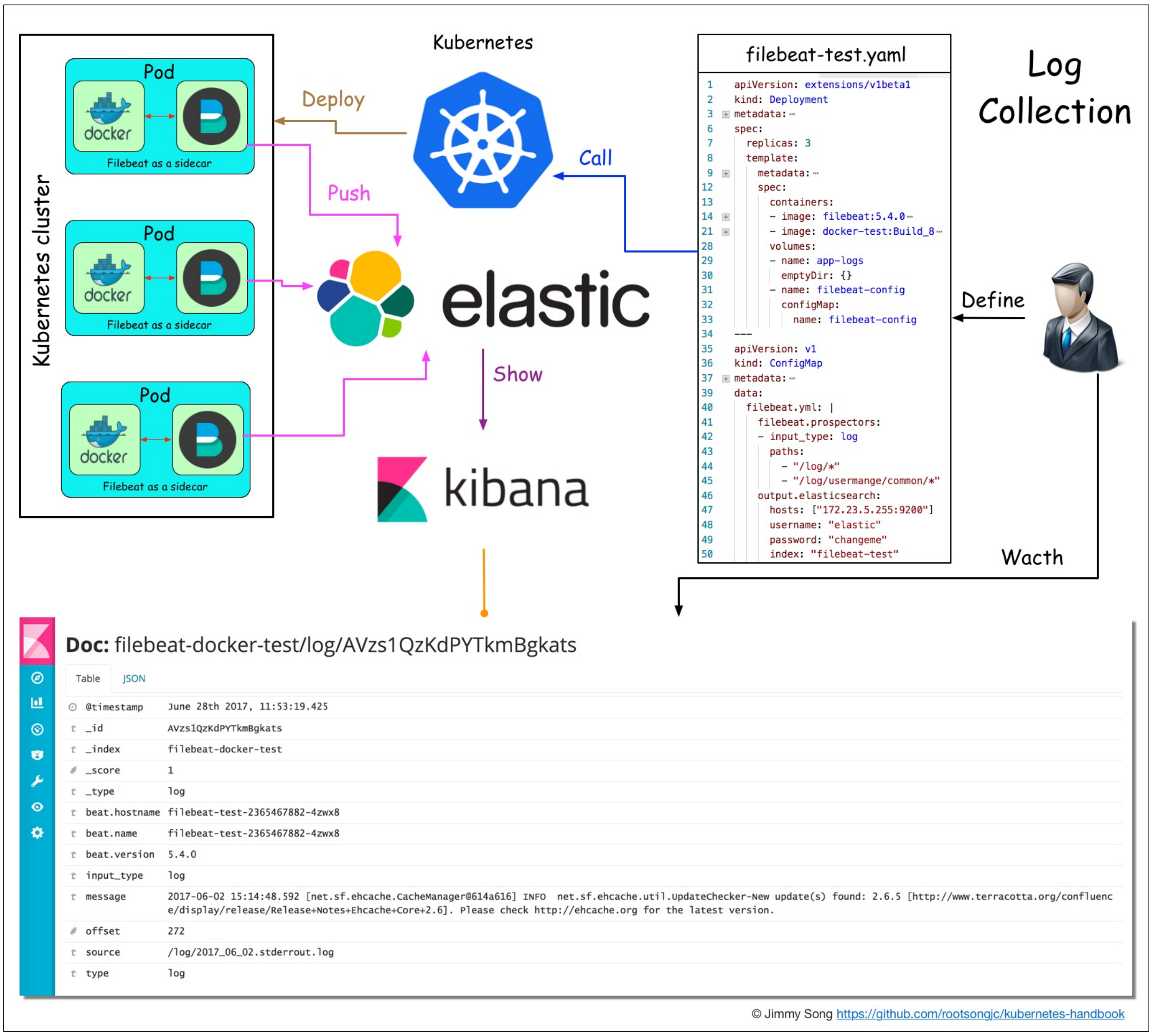
基于以上几个原因，我们决定使用自己的ELK集群。

Kubernetes集群中的日志收集解决方案

编号	方案	优点	缺点
1	每个app的镜像中都集成日志收集组件	部署方便，kubernetes的yaml文件无须特别配置，可以为每个app自定义日志收集配置	强耦合，不方便应用和日志收集组件升级和维护且会导致镜像过大
2	单独创建一个日志收集组件跟app的容器一起运行在同一个pod中	低耦合，扩展性强，方便维护和升级	需要对kubernetes的yaml文件进行单独配置，略显繁琐
3	将所有的Pod的日志都挂载到宿主机上，每台主机上单独起一个日志收集Pod	完全解耦，性能最高，管理起来最方便	需要统一日志收集规则，目录和输出方式

综合以上优缺点，我们选择使用方案二。

该方案在扩展性、个性化、部署和后期维护方面都能做到均衡，因此选择该方案。



图片 - *filebeat*日志收集架构图

我们创建了自己的filebeat镜像。创建过程和使用方式见 https://github.com/rootsongjc/docker-images

镜像地址：`index.tenxcloud.com/jimmy/filebeat:5.4.0`

测试

我们部署一个应用filebeat来收集日志的功能测试。

创建应用yaml文件 `filebeat-test.yaml` 。

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
```

```
  name: filebeat-test
  namespace: default
spec:
  replicas: 3
  template:
    metadata:
      labels:
        k8s-app: filebeat-test
    spec:
      containers:
      - image: harbor-001.jimmysong.io/library/filebeat:5.4.0
        name: filebeat
        volumeMounts:
        - name: app-logs
          mountPath: /log
        - name: filebeat-config
          mountPath: /etc/filebeat/
      - image: harbor-001.jimmysong.io/library/analytics-docker-
test:Build_8
        name : app
        ports:
        - containerPort: 80
        volumeMounts:
        - name: app-logs
          mountPath: /usr/local/TalkingData/logs
      volumes:
      - name: app-logs
        emptyDir: {}
      - name: filebeat-config
        configMap:
          name: filebeat-config
---
apiVersion: v1
kind: Service
metadata:
  name: filebeat-test
  labels:
    app: filebeat-test
spec:
  ports:
  - port: 80
    protocol: TCP
    name: http
  selector:
    run: filebeat-test
---
apiVersion: v1
kind: ConfigMap
metadata:
  name: filebeat-config
data:
  filebeat.yml: |
    filebeat.prospectors:
```

```
- input_type: log
  paths:
    - "/log/*"
    - "/log/usermange/common/*"
output.elasticsearch:
  hosts: ["172.23.5.255:9200"]
  username: "elastic"
  password: "changeme"
  index: "filebeat-docker-test"
```

说明

该文件中包含了配置文件filebeat的配置文件的ConfigMap，因此不需要再定义环境变量。

当然你也可以不同ConfigMap，通过传统的传递环境变量的方式来配置filebeat。

例如对filebeat的容器进行如下配置：

```
containers:
- image: harbor-001.jimmysong.io/library/filebeat:5.4.0
  name: filebeat
  volumeMounts:
  - name: app-logs
    mountPath: /log
  env:
  - name: PATHS
    value: "/log/*"
  - name: ES_SERVER
    value: 172.23.5.255:9200
  - name: INDEX
    value: logstash-docker
  - name: INPUT_TYPE
    value: log
```

目前使用这种方式会有个问题，及时 `PATHS` 只能传递单个目录，如果想传递多个目录需要修改filebeat镜像的 `docker-entrypoint.sh` 脚本，对该环境变量进行解析增加filebeat.yml文件中的PATHS列表。

推荐使用ConfigMap，这样filebeat的配置就能够更灵活。

注意事项

- 将app的 `/usr/local/TalkingData/logs` 目录挂载到filebeat的 `/log` 目录下。
- 该文件可以在 `manifests/test/filebeat-test.yaml` 找到。

- 我使用了自己的私有镜像仓库，测试时请换成自己的应用镜像。
- Filebeat的环境变量的值配置请参考https://github.com/rootsongjc/docker-images

创建应用

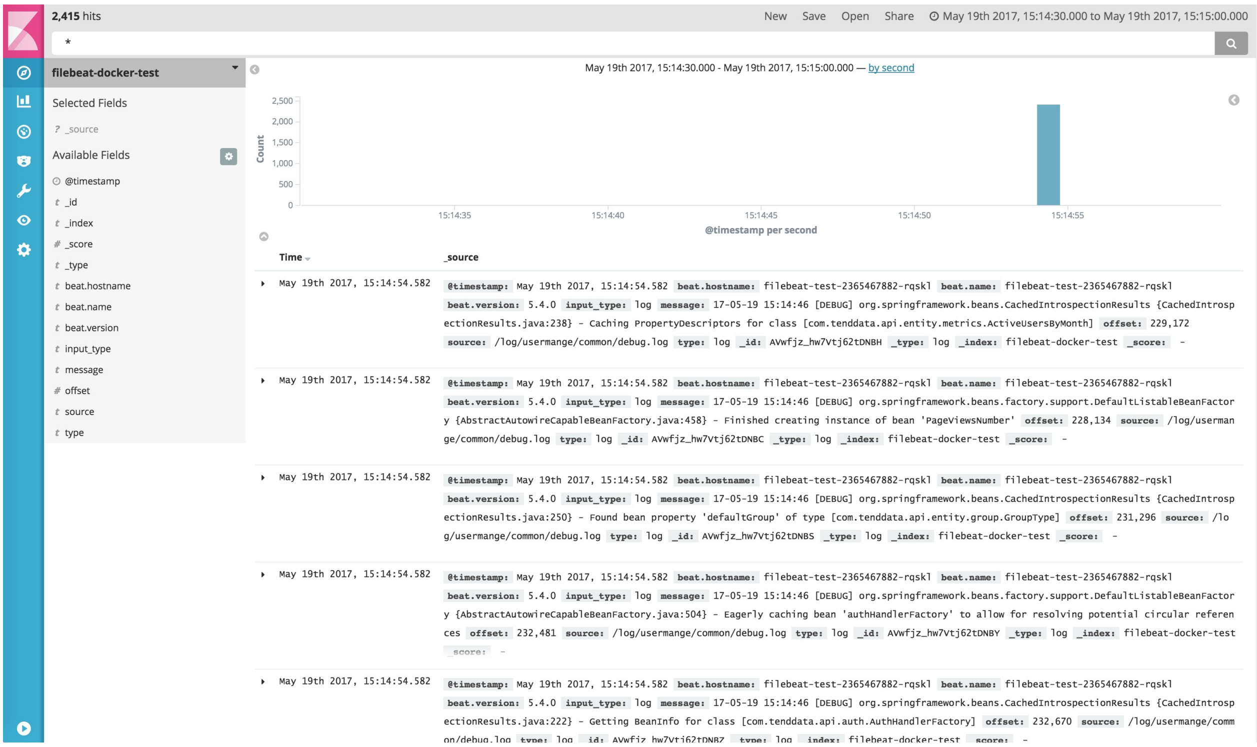
部署Deployment

```
kubectl create -f filebeat-test.yaml
```

查看 `http://172.23.5.255:9200/_cat/indices` 将可以看到列表有这样的indices：

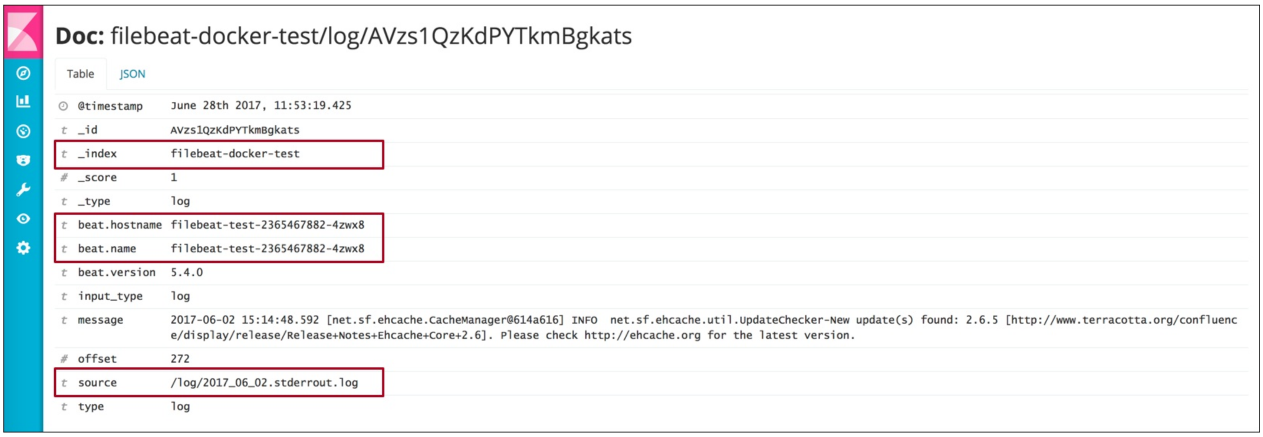
```
green open filebeat-docker-test            7xPEwEbUQRirk8oDX36gA
A 5 1   2151      0   1.6mb 841.8kb
```

访问Kibana的web页面，查看 `filebeat-2017.05.17` 的索引，可以看到filebeat收集到了app日志。



图片 - *Kibana*页面

点开每个日志条目，可以看到以下详细字段：



图片 - *filebeat*收集的日志详细信息

- `_index` 值即我们在YAML文件的 `configMap` 中配置的index值
- `beat.hostname` 和 `beat.name` 即pod的名称
- source表示filebeat容器中的日志目录

我们可以通过人为得使 `index` = `service name` ，这样就可以方便的收集和查看每个service的日志。

配置最佳实践

本文档旨在汇总和强调用户指南、快速开始文档和示例中的最佳实践。该文档会很活跃并持续更新中。如果你觉得很有用的最佳实践但是本文档中没有包含，欢迎给我们提Pull Request。

通用配置建议

- 定义配置文件的时候，指定最新的稳定API版本（目前是V1）。
- 在配置文件push到集群之前应该保存在版本控制系统中。这样当需要的时候能够快速回滚，必要的时候也可以快速的创建集群。
- 使用YAML格式而不是JSON格式的配置文件。在大多数场景下它们都可以作为数据交换格式，但是YAML格式比起JSON更易读和配置。
- 尽量将相关的对象放在同一个配置文件里。这样比分成多个文件更容易管理。参考guestbook-all-in-one.yaml文件中的配置（注意，尽管你可以在使用 `kubectl` 命令时指定配置文件目录，你也可以在配置文件目录下执行 `kubectl create` ——查看下面的详细信息）。
- 为了简化和最小化配置，也为了防止错误发生，不要指定不必要的默认配置。例如，省略掉 `ReplicationController` 的selector和label，如果你希望它们跟 `podTemplate` 中的label一样的话，因为那些配置默认是 `podTemplate` 的label产生的。更多信息请查看 guestbook app 的yaml文件和 examples 。
- 将资源对象的描述放在一个annotation中可以更好的内省。

裸的Pods vs Replication Controllers和 Jobs

- 如果有其他方式替代"裸的" pod（如没有绑定到replication controller 上的pod），那么就使用其他选择。在node节点出现故障时，裸奔的pod不会被重新调度。Replication Controller总是会重新创建pod，除了明确指定了 `restartPolicy: Never` 的场景。Job 也许是比较合适的选择。

Services

- 通常最好在创建相关的replication controllers之前先创建service ，你也可以在

创建Replication Controller的时候不指定replica数量（默认是1），创建service后，在通过Replication Controller来扩容。这样可以在扩容很多个replica之前先确认pod是正常的。

- 除非十分必要的情况下（如运行一个node daemon），不要使用 `hostPort` （用来指定暴露在主机上的端口号）。当你给Pod绑定了一个 `hostPort` ，该pod可被调度到的主机的受限了，因为端口冲突。如果是为了调试目的来通过端口访问的话，你可以使用 kubectl proxy and apiserver proxy 或者 kubectl port-forward。你可使用 Service 来对外暴露服务。如果你确实需要将pod的端口暴露到主机上，考虑使用 NodePort service。
- 跟 `hostPort` 一样的原因，避免使用 `hostNetwork` 。
- 如果你不需要kube-proxy的负载均衡的话，可以考虑使用使用headless services。

使用Label

- 定义 labels 来指定应用或Deployment的 **semantic attributes** 。例如，不是将label附加到一组pod来显式表示某些服务（例如， `service:myservice` ），或者显式地表示管理pod的replication controller（例如， `controller:mycontroller` ），附加label应该是标示语义属性的标签， 例如 `{app:myapp,tier:frontend,phase:test,deployment:v3}` 。 这将允许您选择适合上下文的对象组——例如，所有的”tier:frontend“pod的服务或app是“myapp”的所有“测试”阶段组件。 有关此方法的示例，请参阅guestbook应用程序。

  可以通过简单地从其service的选择器中省略特定于发行版本的标签，而不是更新服务的选择器来完全匹配replication controller的选择器，来实现跨越多个部署的服务，例如滚动更新。

- 为了滚动升级的方便，在Replication Controller的名字中包含版本信息，例如作为名字的后缀。设置一个 `version` 标签页是很有用的。滚动更新创建一个新的controller而不是修改现有的controller。因此，version含混不清的controller名字就可能带来问题。查看Rolling Update Replication Controller文档获取更多关于滚动升级命令的信息。

  注意 Deployment 对象不需要再管理 replication controller 的版本名。Deployment 中描述了对象的期望状态，如果对spec的更改被应用了话，Deployment controller 会以控制的速率来更改实际状态到期望状态。

（Deployment目前是 `extensions` API Group的一部分）。

- 利用label做调试。因为Kubernetes replication controller和service使用label来匹配pods，这允许你通过移除pod中的label的方式将其从一个controller或者service中移除，原来的controller会创建一个新的pod来取代移除的pod。这是一个很有用的方式，帮你在一个隔离的环境中调试之前的“活着的” pod。查看 `kubectl label` 命令。

容器镜像

- 默认容器镜像拉取策略 是 `IfNotPresent` , 当本地已存在该镜像的时候 Kubelet 不会再从镜像仓库拉取。如果你希望总是从镜像仓库中拉取镜像的话，在yaml文件中指定镜像拉取策略为 `Always` （ `imagePullPolicy: Always` ）或者指定镜像的tag为 `:latest` 。

  如果你没有将镜像标签指定为 `:latest` ，例如指定为 `myimage:v1` ，当该标签的镜像进行了更新，kubelet也不会拉取该镜像。你可以在每次镜像更新后都生成一个新的tag（例如 `myimage:v2` ），在配置文件中明确指定该版本。

  注意： 在生产环境下部署容器应该尽量避免使用 `:latest` 标签，因为这样很难追溯到底运行的是哪个版本的容器和回滚。

使用kubectl

- 尽量使用 `kubectl create -f <directory>` 。kubeclt会自动查找该目录下的所有后缀名为 `.yaml` 、 `.yml` 和 `.json` 文件并将它们传递给 `create` 命令。
- 使用 `kubectl delete` 而不是 `stop` . `Delete` 是 `stop` 的超集， `stop` 已经被弃用。
- 使用 kubectl bulk 操作（通过文件或者label）来get和delete。查看label selectors 和 using labels effectively。
- 使用 `kubectl run` 和 `expose` 命令快速创建只有单个容器的Deployment。查看 quick start guide中的示例。

参考

- Configuration Best Practices

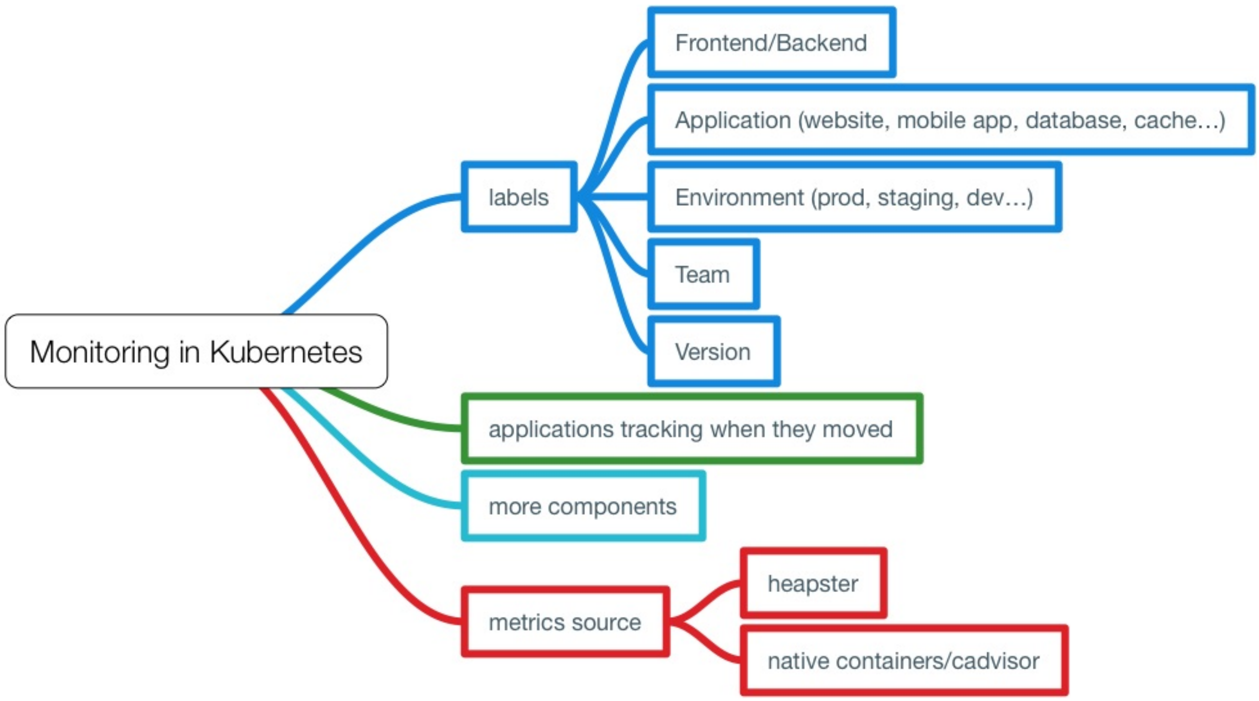
集群及应用监控

在前面的安装heapster插件章节，我们已经谈到Kubernetes本身提供了监控插件作为集群和容器监控的选择，但是在实际使用中，因为种种原因，再考虑到跟我们自身的监控系统集成，我们准备重新造轮子。

针对kubernetes集群和应用的监控，相较于传统的虚拟机和物理机的监控有很多不同，因此对于传统监控需要有很多改造的地方，需要关注以下三个方面：

- Kubernetes集群本身的监控，主要是kubernetes的各个组件
- kubernetes集群中Pod的监控，Pod的CPU、内存、网络、磁盘等监控
- 集群内部应用的监控，针对应用本身的监控

Kubernetes集群中的监控



图片 - *Kubernetes*集群中的监控

跟物理机器和虚拟机的监控不同，在kubernetes集群中的监控复杂度更高一些，因为多了一个虚拟化层，当然这个跟直接监控docker容器又不一样，kubernetes在docker之上又抽象了一层service的概念。

在kubernetes中的监控需要考虑到这几个方面：

- 应该给Pod打上哪些label，这些label将成为监控的metrics。
- 当应用的Pod漂移了之后怎么办？因为要考虑到Pod的生命周期比虚拟机和物理机短的多，如何持续监控应用的状态？
- 更多的监控项，kubernetes本身、容器、应用等。
- 监控指标的来源，是通过heapster收集后汇聚还是直接从每台主机的docker上取？

容器的命名规则

首先我们需要清楚使用cAdvisor收集的数据的格式和字段信息。

当我们通过cAdvisor获取到了容器的信息后，例如访问 `${NODE_IP}:4194/api/v1.3/docker` 获取的json结果中的某个容器包含如下字段：

```
        "labels": {
            "annotation.io.kubernetes.container.hash": "f47f0602"
,
            "annotation.io.kubernetes.container.ports": "[{\"containerPort\":80,\"protocol\":\"TCP\"}]",
            "annotation.io.kubernetes.container.restartCount": "0",
            "annotation.io.kubernetes.container.terminationMessagePath": "/dev/termination-log",
            "annotation.io.kubernetes.container.terminationMessagePolicy": "File",
            "annotation.io.kubernetes.pod.terminationGracePeriod": "30",
            "io.kubernetes.container.logpath": "/var/log/pods/d8a2e995-3617-11e7-a4b0-ecf4bbe5d414/php-redis_0.log",
            "io.kubernetes.container.name": "php-redis",
            "io.kubernetes.docker.type": "container",
            "io.kubernetes.pod.name": "frontend-2337258262-771lz"
,
            "io.kubernetes.pod.namespace": "default",
            "io.kubernetes.pod.uid": "d8a2e995-3617-11e7-a4b0-ecf4bbe5d414",
            "io.kubernetes.sandbox.id": "843a0f018c0cef2a5451434713ea3f409f0debc2101d2264227e814ca0745677"
        },
```

这些信息其实都是kubernetes创建容器时给docker container打的 `Labels` ，使用 `docker inspect $conainer_name` 命令同样可以看到上述信息。

你是否想过这些label跟容器的名字有什么关系？当你在node节点上执行 `docker ps` 看到的容器名字又对应哪个应用的Pod呢？

在kubernetes代码中pkg/kubelet/dockertools/docker.go中的BuildDockerName方法定义了容器的名称规范。

这段容器名称定义代码如下：

```
// Creates a name which can be reversed to identify both full po
d name and container name.
// This function returns stable name, unique name and a unique i
d.
// Although rand.Uint32() is not really unique, but it's enough
for us because error will
// only occur when instances of the same container in the same p
od have the same UID. The
// chance is really slim.
func BuildDockerName(dockerName KubeletContainerName, container *
v1.Container) (string, string, string) {
	containerName := dockerName.ContainerName + "." + strconv.Fo
rmatUint(kubecontainer.HashContainerLegacy(container), 16)
	stableName := fmt.Sprintf("%s_%s_%s_%s",
		containerNamePrefix,
		containerName,
		dockerName.PodFullName,
		dockerName.PodUID)
	UID := fmt.Sprintf("%08x", rand.Uint32())
	return stableName, fmt.Sprintf("%s_%s", stableName, UID), UI
D
}

// Unpacks a container name, returning the pod full name and con
tainer name we would have used to
// construct the docker name. If we are unable to parse the name
, an error is returned.
func ParseDockerName(name string) (dockerName *KubeletContainerN
ame, hash uint64, err error) {
	// For some reason docker appears to be appending '/' to nam
es.
	// If it's there, strip it.
	name = strings.TrimPrefix(name, "/")
	parts := strings.Split(name, "_")
	if len(parts) == 0 || parts[0] != containerNamePrefix {
		err = fmt.Errorf("failed to parse Docker container name
%q into parts", name)
		return nil, 0, err
	}
```

```
    if len(parts) < 6 {
        // We have at least 5 fields.  We may have more in the f
uture.
        // Anything with less fields than this is not something
we can
        // manage.
        glog.Warningf("found a container with the %q prefix, but
 too few fields (%d): %q", containerNamePrefix, len(parts), name)

        err = fmt.Errorf("Docker container name %q has less part
s than expected %v", name, parts)
        return nil, 0, err
    }

    nameParts := strings.Split(parts[1], ".")
    containerName := nameParts[0]
    if len(nameParts) > 1 {
        hash, err = strconv.ParseUint(nameParts[1], 16, 32)
        if err != nil {
            glog.Warningf("invalid container hash %q in containe
r %q", nameParts[1], name)
        }
    }

    podFullName := parts[2] + "_" + parts[3]
    podUID := types.UID(parts[4])

    return &KubeletContainerName{podFullName, podUID, containerN
ame}, hash, nil
}
```

我们可以看到容器名称中包含如下几个字段，中间用下划线隔开，至少有6个字段，未来可能添加更多字段。

下面的是四个基本字段。

```
containerNamePrefix_containerName_PodFullName_PodUID
```

所有kubernetes启动的容器的containerNamePrefix都是k8s。

Kubernetes启动的docker容器的容器名称规范，下面以官方示例guestbook为例，Deployment 名为 frontend中启动的名为php-redis的docker容器的副本书为3。

Deployment frontend的配置如下：

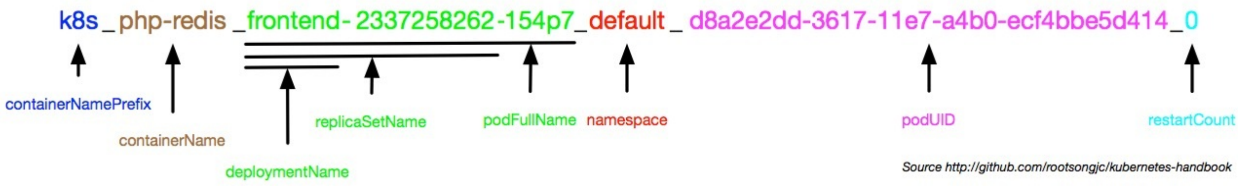
```yaml
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: frontend
spec:
  template:
    metadata:
      labels:
        app: guestbook
        tier: frontend
    spec:
      containers:
      - name: php-redis
        image: harbor-001.jimmysong.io/library/gb-frontend:v4
        resources:
          requests:
            cpu: 100m
            memory: 100Mi
        env:
        - name: GET_HOSTS_FROM
          value: dns
        ports:
        - containerPort: 80
```

我们选取三个实例中的一个运行php-redis的docker容器。

```
k8s_php-redis_frontend-2337258262-154p7_default_d8a2e2dd-3617-11
e7-a4b0-ecf4bbe5d414_0
```

- containerNamePrefix：k8s
- containerName：php-redis
- podFullName：frontend-2337258262-154p7
- computeHash：154p7
- deploymentName：frontend
- replicaSetName：frontend-2337258262
- namespace：default
- podUID：d8a2e2dd-3617-11e7-a4b0-ecf4bbe5d414

kubernetes容器命名规则解析，见下图所示。

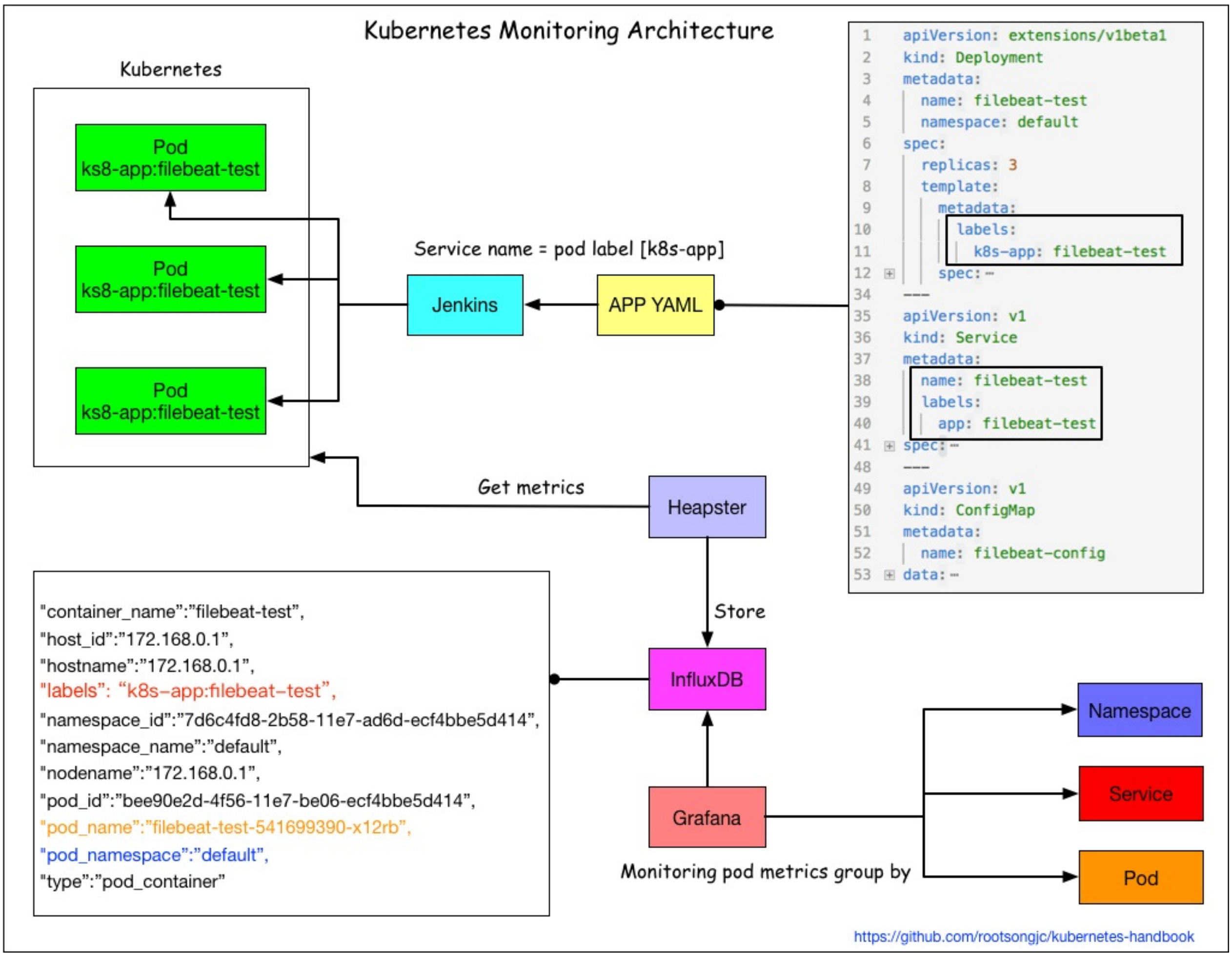


图片 - *kubernetes*的容器命名规则示意图

使用Heapster进行集群监控

Heapster是kubernetes官方提供的监控方案，我们在前面的章节中已经讲解了如何部署和使用heapster，见安装Heapster插件。

但是Grafana显示的指标只根据Namespace和Pod两层来分类，实在有些单薄，我们希望通过应用的label增加service这一层分类。架构图如下：

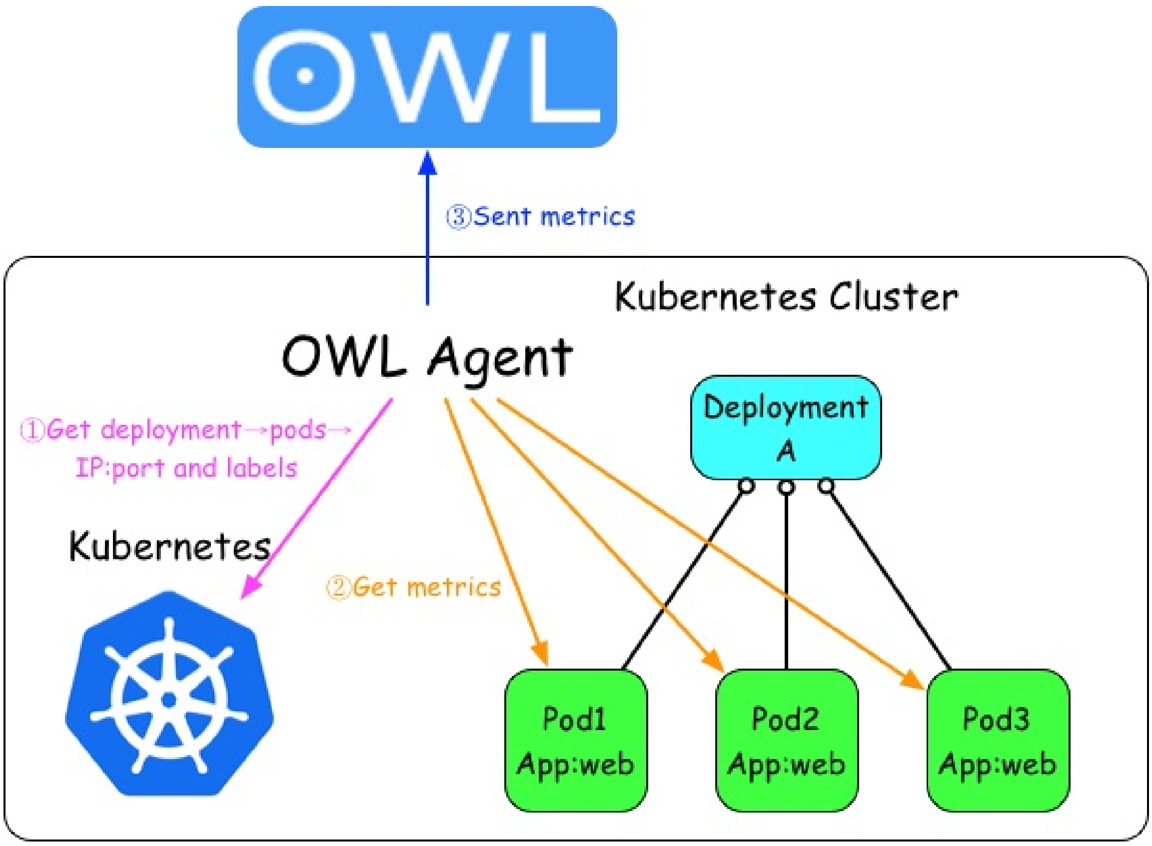


图片 - *Heapster*架构图（改进版）

在不改变原有架构的基础上，通过应用的label来区分不同应用的pod。

应用监控

Kubernetes中应用的监控架构如图：



图片 - 应用监控架构图

这种方式有以下几个要点：

- 访问kubernetes API获取应用Pod的IP和端口
- Pod labels作为监控metric的tag
- 直接访问应用的Pod的IP和端口获取应用监控数据
- metrics发送到OWL中存储和展示

应用拓扑状态图

对于复杂的应用编排和依赖关系，我们希望能够有清晰的图标一览应用状态和拓扑关系，因此我们用到了Weaveworks开源的scope。

安装scope

我们在kubernetes集群上使用standalone方式安装，详情参考Installing Weave Scope。

使用scope.yaml文件安装scope，该服务安装在 `kube-system` namespace下。

```
$ kubectl apply -f scope.yaml
```

创建一个新的Ingress： `kube-system.yaml` ，配置如下：

```yaml
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: traefik-ingress
  namespace: kube-system
spec:
  rules:
    - host: scope.weave.io
      http:
        paths:
        - path: /
          backend:
            serviceName: weave-scope-app
            servicePort: 80
```

执行 `kubectl apply -f kube-system.yaml` 后在你的主机上的 `/etc/hosts` 文件中添加一条记录：

```
172.20.0.119 scope.weave.io
```

在浏览器中访问 `scope.weave.io` 就可以访问到scope了，详见边缘节点配置。



图片 - 应用拓扑图

如上图所示，scope可以监控kubernetes集群中的一系列资源的状态、资源使用情况、应用拓扑、scale、还可以直接通过浏览器进入容器内部调试等。

参考

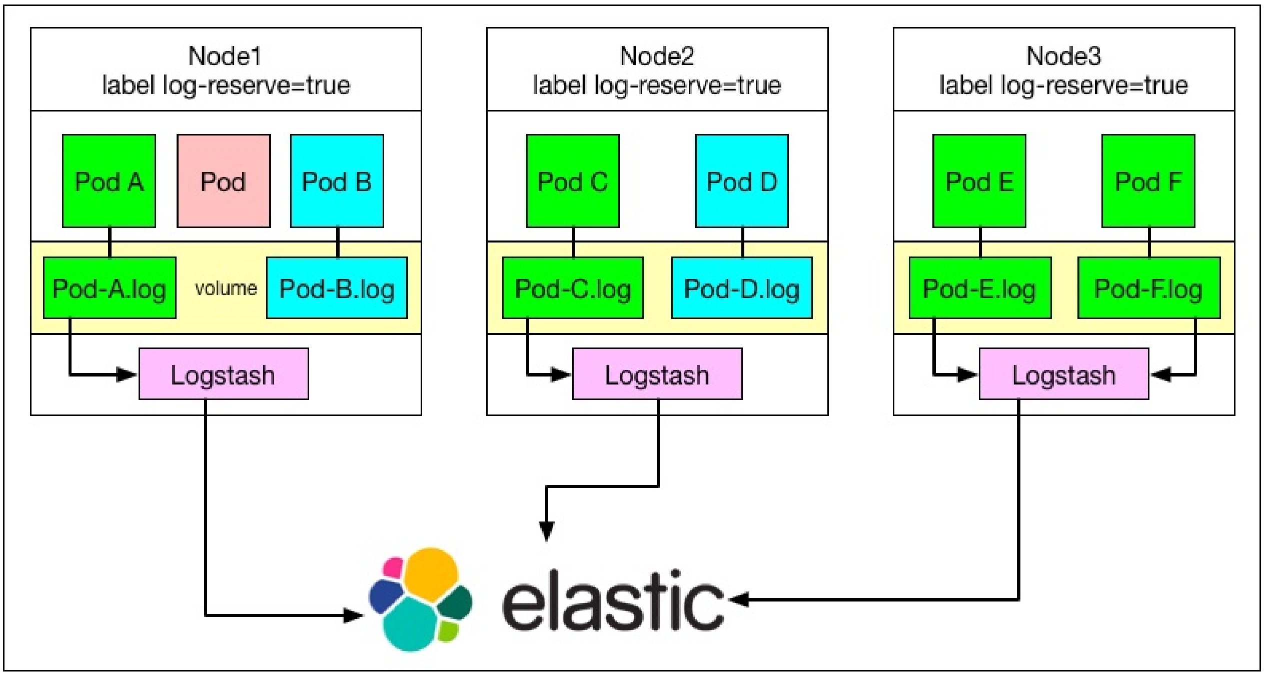
- Monitoring in the Kubernetes Era

数据落盘问题的由来

这本质上是数据持久化问题，对于有些应用依赖持久化数据，比如应用自身产生的日志需要持久化存储的情况，需要保证容器里的数据不丢失，在Pod挂掉后，其他应用依然可以访问到这些数据，因此我们需要将数据持久化存储起来。

数据落盘问题解决方案

下面以一个应用的日志收集为例，该日志需要持久化收集到ElasticSearch集群中，如果不考虑数据丢失的情形，可以直接使用前面提到的应用日志收集一节中的方法，但考虑到Pod挂掉时logstash（或filebeat）并没有收集完该pod内日志的情形，我们想到了如下这种解决方案，示意图如下：



图片 - 日志持久化收集解决方案示意图

1. 首先需要给数据落盘的应用划分node，即这些应用只调用到若干台主机上
2. 给这若干台主机增加label
3. 使用 `deamonset` 方式在这若干台主机上启动logstash的Pod（使用nodeSelector来限定在这几台主机上，我们在边缘节点启动的 `treafik` 也是这种模式）

4. 将应用的数据通过volume挂载到宿主机上
5. Logstash（或者filebeat）收集宿主机上的数据，数据持久化不会丢失

Side-effect

1. 首先kubernetes本身就提供了数据持久化的解决方案statefulset，不过需要用到公有云的存储货其他分布式存储，这一点在我们的私有云环境里被否定了。
2. 需要管理主机的label，增加运维复杂度，但是具体问题具体对待
3. 必须保证应用启动顺序，需要先启动logstash
4. 为主机打label使用nodeSelector的方式限制了资源调度的范围

管理容器的计算资源

当您定义 Pod 的时候可以选择为每个容器指定需要的 CPU 和内存（RAM）大小。当为容器指定了资源请求后，调度器就能够更好的判断出将容器调度到哪个节点上。如果您还为容器指定了资源限制，节点上的资源就可以按照指定的方式做竞争。关于资源请求和限制的不同点和更多资料请参考 Resource QoS。

资源类型

CPU 和 *memory* 都是 资源类型。资源类型具有基本单位。CPU 的单位是 core，memory 的单位是 byte。

CPU和内存统称为计算资源，也可以称为资源。计算资源的数量是可以被请求、分配和消耗的可测量的。它们与 API 资源 不同。 API 资源（如 Pod 和 Service）是可通过 Kubernetes API server 读取和修改的对象。

Pod 和 容器的资源请求和限制

Pod 中的每个容器都可以指定以下的一个或者多个值：

- `spec.containers[].resources.limits.cpu`
- `spec.containers[].resources.limits.memory`
- `spec.containers[].resources.requests.cpu`
- `spec.containers[].resources.requests.memory`

尽管只能在个别容器上指定请求和限制，但是我们可以方便地计算出 Pod 资源请求和限制。特定资源类型的Pod 资源请求/限制是 Pod 中每个容器的该类型的资源请求/限制的总和。

CPU 的含义

CPU 资源的限制和请求以 *cpu* 为单位。

Kubernetes 中的一个 cpu 等于：

- 1 AWS vCPU

- 1 GCP Core
- 1 Azure vCore
- 1 *Hyperthread* 在带有超线程的裸机 Intel 处理器上

允许浮点数请求。具有 `spec.containers[].resources.requests.cpu` 为 0.5 的容器保证了一半 CPU 要求 1 CPU的一半。表达式 `0.1` 等价于表达式 `100m` ，可以看作 “100 millicpu”。有些人说成是“一百毫 cpu”，其实说的是同样的事情。具有小数点（如 `0.1` ）的请求由 API 转换为 `100m` ，精度不超过 `1m` 。因此，可能会优先选择 `100m` 的形式。

CPU 总是要用绝对数量，不可以使用相对数量；0.1 的 CPU 在单核、双核、48核的机器中的意义是一样的。

内存的含义

内存的限制和请求以字节为单位。您可以使用以下后缀之一作为平均整数或定点整数表示内存：E，P，T，G，M，K。您还可以使用两个字母的等效的幂数：Ei，Pi，Ti ，Gi，Mi，Ki。例如，以下代表大致相同的值：

```
128974848, 129e6, 129M, 123Mi
```

下面是个例子。

以下 Pod 有两个容器。每个容器的请求为 0.25 cpu 和 64MiB（2^{26} 字节）内存，每个容器的限制为 0.5 cpu 和 128MiB 内存。您可以说该 Pod 请求 0.5 cpu 和 128 MiB 的内存，限制为 1 cpu 和 256MiB 的内存。

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: frontend
spec:
  containers:
  - name: db
    image: mysql
    resources:
      requests:
        memory: "64Mi"
        cpu: "250m"
      limits:
        memory: "128Mi"
        cpu: "500m"
  - name: wp
    image: wordpress
    resources:
      requests:
        memory: "64Mi"
        cpu: "250m"
      limits:
        memory: "128Mi"
        cpu: "500m"
```

具有资源请求的 Pod 如何调度

当您创建一个 Pod 时，Kubernetes 调度程序将为 Pod 选择一个节点。每个节点具有每种资源类型的最大容量：可为 Pod 提供的 CPU 和内存量。调度程序确保对于每种资源类型，调度的容器的资源请求的总和小于节点的容量。请注意，尽管节点上的实际内存或 CPU 资源使用量非常低，但如果容量检查失败，则调度程序仍然拒绝在该节点上放置 Pod。当资源使用量稍后增加时，例如在请求率的每日峰值期间，这可以防止节点上的资源短缺。

具有资源限制的 Pod 如何运行

当 kubelet 启动一个 Pod 的容器时，它会将 CPU 和内存限制传递到容器运行时。

当使用 Docker 时：

- `spec.containers[].resources.requests.cpu` 的值将转换成 millicore 值，这是个浮点数，并乘以1024，这个数字中的较大者或2用作 `docker run` 命令中的 `--cpu-shares` 标志的值。

- `spec.containers[].resources.limits.cpu` 被转换成 millicore 值。被乘以 100000 然后 除以 1000。这个数字用作 `docker run` 命令中的 `--cpu-quota` 标志的值。[`--cpu-quota`] 标志被设置成了 100000，表示测量配额使用的默认100ms 周期。如果 [`--cpu-cfs-quota`] 标志设置为 true，则 kubelet 会强制执行 cpu 限制。从 Kubernetes 1.2 版本起，此标志默认为 true。
- `spec.containers[].resources.limits.memory` 被转换为整型，作为 `docker run` 命令中的 `--memory` 标志的值。

如果容器超过其内存限制，则可能会被终止。如果可重新启动，则与所有其他类型的运行时故障一样，kubelet 将重新启动它。

如果一个容器超过其内存请求，那么当节点内存不足时，它的 Pod 可能被逐出。

容器可能被允许也可能不被允许超过其 CPU 限制时间。但是，由于 CPU 使用率过高，不会被杀死。

要确定容器是否由于资源限制而无法安排或被杀死，请参阅 疑难解答 部分。

监控计算资源使用

Pod 的资源使用情况被报告为 Pod 状态的一部分。

如果为集群配置了 可选监控，则可以从监控系统检索 Pod 资源的使用情况。

疑难解答

我的 Pod 处于 pending 状态且事件信息显示 failedScheduling

如果调度器找不到任何该 Pod 可以匹配的节点，则该 Pod 将保持不可调度状态，直到找到一个可以被调度到的位置。每当调度器找不到 Pod 可以调度的地方时，会产生一个事件，如下所示：

```
$ kubectl describe pod frontend | grep -A 3 Events
Events:
  FirstSeen LastSeen   Count  From          Subobject   PathReas
on      Message
  36s   5s      6      {scheduler }              FailedScheduling
  Failed for reason PodExceedsFreeCPU and possibly others
```

在上述示例中，由于节点上的 CPU 资源不足，名为 “frontend” 的 Pod 将无法调度。由于内存不足（PodExceedsFreeMemory），类似的错误消息也可能会导致失败。一般来说，如果有这种类型的消息而处于 pending 状态，您可以尝试如下几件事情：

```
$ kubectl describe nodes e2e-test-minion-group-4lw4
Name:            e2e-test-minion-group-4lw4
[ ... lines removed for clarity ...]
Capacity:
 alpha.kubernetes.io/nvidia-gpu:    0
 cpu:                               2
 memory:                            7679792Ki
 pods:                              110
Allocatable:
 alpha.kubernetes.io/nvidia-gpu:    0
 cpu:                               1800m
 memory:                            7474992Ki
 pods:                              110
[ ... lines removed for clarity ...]
Non-terminated Pods:        (5 in total)
  Namespace    Name                                  CPU Request
s  CPU Limits  Memory Requests  Memory Limits
  ---------    ----                                  -----------
-  ----------  ---------------  -------------
  kube-system  fluentd-gcp-v1.38-28bv1               100m (5%)
   0 (0%)      200Mi (2%)       200Mi (2%)
  kube-system  kube-dns-3297075139-61lj3             260m (13%)
   0 (0%)      100Mi (1%)       170Mi (2%)
  kube-system  kube-proxy-e2e-test-...               100m (5%)
   0 (0%)      0 (0%)           0 (0%)
  kube-system  monitoring-influxdb-grafana-v4-z1m12  200m (10%)
   200m (10%)  600Mi (8%)       600Mi (8%)
  kube-system  node-problem-detector-v0.1-fj7m3      20m (1%)
   200m (10%)  20Mi (0%)        100Mi (1%)
Allocated resources:
  (Total limits may be over 100 percent, i.e., overcommitted.)
  CPU Requests    CPU Limits    Memory Requests    Memory Limits
  ------------    ----------    ---------------    -------------
  680m (34%)      400m (20%)    920Mi (12%)        1070Mi (14%)
```

我的容器被终结了

您的容器可能因为资源枯竭而被终结了。要查看容器是否因为遇到资源限制而被杀死，请在相关的 Pod 上调用 `kubectl describe pod` ：

```
[12:54:41] $ kubectl describe pod simmemleak-hra99
Name:                           simmemleak-hra99
Namespace:                      default
Image(s):                       saadali/simmemleak
Node:                           kubernetes-node-tf0f/10.240.216.
66
Labels:                         name=simmemleak
Status:                         Running
Reason:
Message:
IP:                             10.244.2.75
Replication Controllers:        simmemleak (1/1 replicas created)

Containers:
  simmemleak:
    Image:  saadali/simmemleak
    Limits:
      cpu:                      100m
      memory:                   50Mi
    State:                      Running
      Started:                  Tue, 07 Jul 2015 12:54:41 -0700
    Last Termination State:     Terminated
      Exit Code:                1
      Started:                  Fri, 07 Jul 2015 12:54:30 -0700
      Finished:                 Fri, 07 Jul 2015 12:54:33 -0700
    Ready:                      False
    Restart Count:              5
Conditions:
  Type      Status
  Ready     False
Events:
  FirstSeen                         LastSeen
     Count  From                              SubobjectPath
                  Reason      Message
  Tue, 07 Jul 2015 12:53:51 -0700   Tue, 07 Jul 2015 12:53:51 -0
700  1      {scheduler }
                  scheduled   Successfully assigned simmemleak-h
ra99 to kubernetes-node-tf0f
  Tue, 07 Jul 2015 12:53:51 -0700   Tue, 07 Jul 2015 12:53:51 -0
700  1      {kubelet kubernetes-node-tf0f}    implicitly require
d container POD   pulled      Pod container image "gcr.io/google
_containers/pause:0.8.0" already present on machine
  Tue, 07 Jul 2015 12:53:51 -0700   Tue, 07 Jul 2015 12:53:51 -0
700  1      {kubelet kubernetes-node-tf0f}    implicitly require
d container POD   created     Created with docker id 6a41280f516
```

```
d
  Tue, 07 Jul 2015 12:53:51 -0700   Tue, 07 Jul 2015 12:53:51 -0
700  1      {kubelet kubernetes-node-tf0f}    implicitly require
d container POD    started     Started with docker id 6a41280f516
d
  Tue, 07 Jul 2015 12:53:51 -0700   Tue, 07 Jul 2015 12:53:51 -0
700  1      {kubelet kubernetes-node-tf0f}    spec.containers{si
mmemleak}          created     Created with docker id 87348f12526
a
```

在上面的例子中， `Restart Count: 5` 意味着 Pod 中的 `simmemleak` 容器被终止并重启了五次。

您可以使用 `kubectl get pod` 命令加上 `-o go-template=...` 选项来获取之前终止容器的状态。

```
[13:59:01] $ kubectl get pod -o go-template='{{range.status.cont
ainerStatuses}}{{"Container Name: "}}{{.name}}{{"\r\nLastState:
"}}{{.lastState}}{{end}}'  simmemleak-60xbc
Container Name: simmemleak
LastState: map[terminated:map[exitCode:137 reason:OOM Killed sta
rtedAt:2015-07-07T20:58:43Z finishedAt:2015-07-07T20:58:43Z cont
ainerID:docker://0e4095bba1feccdfe7ef9fb6ebffe972b4b14285d5acdec
6f0d3ae8a22fad8b2]]
```

您可以看到容器因为 `reason:OOM killed` 被终止， `OOM` 表示 Out Of Memory。

不透明整型资源（Alpha功能）

Kubernetes 1.5 版本中引入不透明整型资源。不透明的整数资源允许集群运维人员发布新的节点级资源，否则系统将不了解这些资源。

用户可以在 Pod 的 spec 中消费这些资源，就像 CPU 和内存一样。调度器负责资源计量，以便在不超过可用量的同时分配给 Pod。

注意： 不透明整型资源在 kubernetes 1.5 中还是 Alpha 版本。只实现了资源计量，节点级别的隔离还处于积极的开发阶段。

不透明整型资源是以 `pod.alpha.kubernetes.io/opaque-int-resource-` 为前缀的资源。API server 将限制这些资源的数量为整数。有效 数量的例子有 `3` 、 `3000m` 和 `3Ki` 。无效数量的例子有 `0.5` 和 `1500m` 。

申请使用不透明整型资源需要两步。首先，集群运维人员必须在一个或多个节点上通告每个节点不透明的资源。然后，用户必须在 Pod 中请求不透明资源。

要发布新的不透明整型资源，集群运维人员应向 API server 提交 `PATCH` HTTP请求，以指定集群中节点的 `status.capacity` 的可用数量。在此操作之后，节点的 `status.capacity` 将包括一个新的资源。 `status.allocatable` 字段由 kubelet 异步地使用新资源自动更新。请注意，由于调度器在评估 Pod 适应度时使用节点 `status.allocatable` 值，所以在使用新资源修补节点容量和请求在该节点上调度资源的第一个 pod 之间可能会有短暂的延迟。

示例

这是一个 HTTP 请求，master 节点是 k8s-master，在 k8s-node-1 节点上通告 5 个 “foo” 资源。

```
PATCH /api/v1/nodes/k8s-node-1/status HTTP/1.1
Accept: application/json
Content-Type: application/json-patch+json
Host: k8s-master:8080

[
  {
    "op": "add",
    "path": "/status/capacity/pod.alpha.kubernetes.io~1opaque-int-resource-foo",
    "value": "5"
  }
]
```

```
curl --header "Content-Type: application/json-patch+json" \
--request PATCH \
--data '[{"op": "add", "path": "/status/capacity/pod.alpha.kubernetes.io~1opaque-int-resource-foo", "value": "5"}]' \
http://k8s-master:8080/api/v1/nodes/k8s-node-1/status
```

注意： 在前面的请求中， `~1` 是 patch 路径中 `/` 字符的编码。JSON-Patch 中的操作路径值被解释为 JSON-Pointer。更多详细信息请参阅 IETF RFC 6901, section 3。

```
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  containers:
  - name: my-container
    image: myimage
    resources:
      requests:
        cpu: 2
        pod.alpha.kubernetes.io/opaque-int-resource-foo: 1
```

计划改进

在 kubernetes 1.5 版本中仅允许在容器上指定资源量。计划改进对所有容器在 Pod 中共享资源的计量，如 emptyDir volume。

在 kubernetes 1.5 版本中仅支持容器对 CPU 和内存的申请和限制。计划增加新的资源类型，包括节点磁盘空间资源和一个可支持自定义 资源类型 的框架。

Kubernetes 通过支持通过多级别的 服务质量 来支持资源的过度使用。

在 kubernetes 1.5 版本中，一个 CPU 单位在不同的云提供商和同一云提供商的不同机器类型中的意味都不同。例如，在 AWS 上，节点的容量报告为 ECU，而在 GCE 中报告为逻辑内核。我们计划修改 cpu 资源的定义，以便在不同的提供商和平台之间保持一致。

集群联邦

Kubernetes v1.9声称单集群最多可支持5000个节点和15万个Pod，我相信很少有公司会部署如此庞大的一个单集群，总有很多情况下因为各种各样的原因我们可能会部署多个集群，但是有时候有想将他们统一起来管理，这时候就需要用到集群联邦（Federation）。

为什么要使用 federation

Federation 使管理多个集群变得简单。它通过提供两个主要构建模块来实现：

- 跨集群同步资源：Federation 提供了在多个集群中保持资源同步的能力。例如，可以保证同一个 deployment 在多个集群中存在。
- 跨集群服务发现：Federation 提供了自动配置 DNS 服务以及在所有集群后端上进行负载均衡的能力。例如，可以提供一个全局 VIP 或者 DNS 记录，通过它可以访问多个集群后端。

Federation 还可以提供一些其它用例：

- 高可用：通过在集群间分布负载并自动配置 DNS 服务和负载均衡，federation 最大限度地减少集群故障的影响。
- 避免厂商锁定：通过更简单的跨集群应用迁移方式，federation 可以防止集群厂商锁定。

Federation 对于单个集群没有用处。基于下面这些原因您可能会需要多个集群：

- 低延迟：通过在多个区域部署集群可以最大限度减少区域近端用户的延迟。
- 故障隔离：拥有多个小集群可能比单个大集群更利于故障隔离（例如：在云服务提供商的不同可用区中的多个集群）。详情请参考 多集群指南。
- 可伸缩性：单个集群有可伸缩性限制（对于大多数用户这不是典型场景。更多细节请参考 Kubernetes 弹性伸缩与性能目标）。
- 混合云：您可以在不同的云服务提供商或本地数据中心中拥有多个集群。

警告

虽然 federation 有很多吸引人的使用案例，但也有一些注意事项：

- 增加网络带宽和成本：federation 控制平面监控所有集群以确保当前状态符合预期。如果集群在云服务提供商的不同区域或者不同的云服务提供商上运行时，这将导致明显的网络成本增加。
- 减少跨集群隔离：federation 控制平面中的 bug 可能影响所有集群。通过在 federation 中实现最少的逻辑可以缓解这种情况。只要有可能，它就将尽力把工作委托给 kubernetes 集群中的控制平面。这种设计和实现在安全性及避免多集群停止运行上也是错误的。
- 成熟度：federation 项目相对比较新，还不是很成熟。并不是所有资源都可用，许多仍然处于 alpha 状态。 Issue 88 列举了团队目前正忙于解决的与系统相关的已知问题。

混合云能力

Kubernetes 集群 federation 可以包含运行在不同云服务提供商（例如 Google Cloud、AWS）及本地（例如在 OpenStack 上）的集群。您只需要按需在适合的云服务提供商和/或地点上简单的创建集群，然后向 Federation API Server 注册每个集群的 API endpoint 和凭据即可。（详细信息请参阅 federation 管理指南）。

此后，您的 API 资源 就可以跨越不同的集群和云服务提供商。

设置 federation

为了能够联合（federate）多个集群，首先需要建立一个 federation 控制平面。请按照 设置指南 设置 federation 控制平面。

API 资源

一旦设置了控制平面，您就可以开始创建 federation API 资源。 以下指南详细介绍了一些资源：

- Cluster
- ConfigMap
- DaemonSets
- Deployment
- Events
- Hpa

- Ingress
- Jobs
- Namespaces
- ReplicaSets
- Secrets
- Services

API 参考文档 列举了 federation apiserver 支持的所有资源。

级联删除

Kubernetes 1.6 版本包括了对联邦资源（federated resources）级联删除的支持。通过级联删除，当您从 federation 控制平面删除一个资源时，会同时删除所有底层集群中对应的资源。

当使用 REST API 时，级联删除没有默认开启。要想在使用 REST API 从 federation 控制平面删除资源时启用级联删除，请设置 `DeleteOptions.orphanDependents=false` 选项。使用 `kubectl delete` 时默认使用级联删除。您可以通过运行 `kubectl delete --cascade=false` 来禁用该功能。

注意：Kubernetes 1.5 版本支持的级联删除仅支持部分 federation 资源。

单个集群范围

在诸如 Google Compute Engine 或者 Amazon Web Services 等 IaaS 服务提供商中，虚拟机存在于 区域 或 可用区 上。我们建议 Kubernetes 集群中的所有虚拟机应该位于相同的可用区，因为：

- 与拥有单个全局 Kubernetes 集群相比，单点故障更少。
- 与跨可用区集群相比，推测单区域集群的可用性属性更容易。
- 当 Kubernetes 开发人员设计系统时（例如对延迟，带宽或相关故障进行假设），它们假设所有的机器都在一个单一的数据中心，或以其它方式紧密连接。

在每个可用区域同时拥有多个集群也是可以的，但总体而言，我们认为少一点更好。 选择较少集群的理由是：

- 某些情况下，在一个集群中拥有更多的节点可以改进 Pod 的装箱打包（较少资源碎片）。
- 减少运维开销（尽管随着运维工具和流程的成熟，优势已经减少）。
- 减少每个集群固定资源成本的开销，例如 apiserver 虚拟机（但对于大中型集群的整体集群成本来说百分比很小）。

拥有多个集群的原因包括：

- 严格的安全策略要求将一类工作与另一类工作隔离开来（但是，请参见下面的分区集群（Partitioning Clusters））。
- 对新的 Kubernetes 发行版或其它集群软件进行灰度测试。

选择正确的集群数量

Kubernetes 集群数量的选择可能是一个相对静态的选择，只是偶尔重新设置。相比之下，依据负载情况和增长，集群的节点数量和 service 的 pod 数量可能会经常变化。

要选择集群的数量，首先要确定使用哪些区域，以便为所有终端用户提供足够低的延迟，以在 Kubernetes 上运行服务（如果您使用内容分发网络（Content Distribution Network，CDN），则 CDN 托管内容的时延要求不需要考虑）。法律问题也可能影响到这一点。例如，拥有全球客户群的公司可能会决定在美国、欧盟、亚太和南亚地区拥有集群。我们将选择的区域数量称为 `R` 。

其次，决定在整体仍然可用的前提下，可以同时有多少集群不可用。将不可用集群的数量称为 `U` 。如果您不能确定，那么 1 是一个不错的选择。

如果在集群故障的情形下允许负载均衡将流量引导到任何区域，则至少需要有比 `R` 或 `U + 1` 数量更大的集群。如果不行的话（例如希望在集群发生故障时对所有用户确保低延迟），那么您需要有数量为 `R * (U + 1)` 的集群（ `R` 个区域，每个中有 `U + 1` 个集群）。无论如何，请尝试将每个集群放在不同的区域中。

最后，如果您的任何集群需要比 Kubernetes 集群最大建议节点数更多的节点，那么您可能需要更多的集群。 Kubernetes v1.3 支持最多 1000 个节点的集群。 Kubernetes v1.8 支持最多 5000 个节点的集群。更多的指导请参见 构建大型集群。

原文地址：https://kubernetes.io/docs/concepts/cluster-administration/federation/

译文地址：https://k8smeetup.github.io/docs/concepts/cluster-administration/federation/

存储管理

GlusterFS

GlusterFS是Scale-Out存储解决方案Gluster的核心，它是一个开源的分布式文件系统，具有强大的横向扩展能力，通过扩展能够支持数PB存储容量和处理数千客户端。GlusterFS借助TCP/IP或InfiniBand RDMA网络将物理分布的存储资源聚集在一起，使用单一全局命名空间来管理数据。GlusterFS基于可堆叠的用户空间设计，可为各种不同的数据负载提供优异的性能。

使用glusterfs做持久化存储

我们复用kubernetes的三台主机做glusterfs存储。

以下步骤参考自：https://www.xf80.com/2017/04/21/kubernetes-glusterfs/（该网站已无法访问）

安装glusterfs

我们直接在物理机上使用yum安装，如果你选择在kubernetes上安装，请参考：https://github.com/gluster/gluster-kubernetes/blob/master/docs/setup-guide.md

```
# 先安装 gluster 源
$ yum install centos-release-gluster -y

# 安装 glusterfs 组件
$ yum install -y glusterfs glusterfs-server glusterfs-fuse glusterfs-rdma glusterfs-geo-replication glusterfs-devel

## 创建 glusterfs 目录
$ mkdir /opt/glusterd

## 修改 glusterd 目录
$ sed -i 's/var\/lib/opt/g' /etc/glusterfs/glusterd.vol

# 启动 glusterfs
$ systemctl start glusterd.service

# 设置开机启动
$ systemctl enable glusterd.service

#查看状态
$ systemctl status glusterd.service
```

配置 glusterfs

```
# 配置 hosts

$ vi /etc/hosts
172.20.0.113   test-001.jimmysong.io
172.20.0.114   test-002.jimmysong.io
172.20.0.115   test-003.jimmysong.io


# 开放端口
$ iptables -I INPUT -p tcp --dport 24007 -j ACCEPT

# 创建存储目录
$ mkdir /opt/gfs_data


# 添加节点到 集群
# 执行操作的本机不需要probe 本机
[root@test-001 ~]#
gluster peer probe test-002.jimmysong.io
gluster peer probe test-003.jimmysong.io

# 查看集群状态
$ gluster peer status
Number of Peers: 2

Hostname: test-002.jimmysong.io
Uuid: f25546cc-2011-457d-ba24-342554b51317
State: Peer in Cluster (Connected)

Hostname: test-003.jimmysong.io
Uuid: 42b6cad1-aa01-46d0-bbba-f7ec6821d66d
State: Peer in Cluster (Connected)
```

配置 volume

GlusterFS中的volume的模式有很多中，包括以下几种：

- 分布卷（默认模式）：即DHT, 也叫 分布卷: 将文件已hash算法随机分布到 一台服务器节点中存储。
- 复制模式：即AFR, 创建volume 时带 replica x 数量: 将文件复制到 replica x 个节点中。
- 条带模式：即Striped, 创建volume 时带 stripe x 数量： 将文件切割成数据块，分别存储到 stripe x 个节点中 (类似raid 0)。
- 分布式条带模式：最少需要4台服务器才能创建。 创建volume 时 stripe 2

server = 4 个节点： 是DHT 与 Striped 的组合型。
- 分布式复制模式：最少需要4台服务器才能创建。 创建volume 时 replica 2 server = 4 个节点：是DHT 与 AFR 的组合型。
- 条带复制卷模式：最少需要4台服务器才能创建。 创建volume 时 stripe 2 replica 2 server = 4 个节点： 是 Striped 与 AFR 的组合型。
- 三种模式混合： 至少需要8台 服务器才能创建。 stripe 2 replica 2 , 每4个节点 组成一个 组。

这几种模式的示例图参考：CentOS7安装GlusterFS。

因为我们只有三台主机，在此我们使用默认的分布卷模式。请勿在生产环境上使用该模式，容易导致数据丢失。

```
# 创建分布卷
$ gluster volume create k8s-volume transport tcp test-001.jimmys
ong.io:/opt/gfs_data test-002.jimmysong.io:/opt/gfs_data test-00
3.jimmysong.io:/opt/gfs_data force

# 查看volume状态
$ gluster volume info
Volume Name: k8s-volume
Type: Distribute
Volume ID: 9a3b0710-4565-4eb7-abae-1d5c8ed625ac
Status: Created
Snapshot Count: 0
Number of Bricks: 3
Transport-type: tcp
Bricks:
Brick1: test-001.jimmysong.io:/opt/gfs_data
Brick2: test-002.jimmysong.io:/opt/gfs_data
Brick3: test-003.jimmysong.io:/opt/gfs_data
Options Reconfigured:
transport.address-family: inet
nfs.disable: on

# 启动 分布卷
$ gluster volume start k8s-volume
```

Glusterfs调优

```
# 开启 指定 volume 的配额
$ gluster volume quota k8s-volume enable

# 限制 指定 volume 的配额
$ gluster volume quota k8s-volume limit-usage / 1TB

# 设置 cache 大小, 默认32MB
$ gluster volume set k8s-volume performance.cache-size 4GB

# 设置 io 线程, 太大会导致进程崩溃
$ gluster volume set k8s-volume performance.io-thread-count 16

# 设置 网络检测时间, 默认42s
$ gluster volume set k8s-volume network.ping-timeout 10

# 设置 写缓冲区的大小, 默认1M
$ gluster volume set k8s-volume performance.write-behind-window-
size 1024MB
```

Kubernetes中配置glusterfs

官方的文档见：https://github.com/kubernetes/kubernetes/tree/master/examples/volumes/glusterfs

以下用到的所有yaml和json配置文件可以在../manifests/glusterfs中找到。注意替换其中私有镜像地址为你自己的镜像地址。

kubernetes安装客户端

```
# 在所有 k8s node 中安装 glusterfs 客户端

$ yum install -y glusterfs glusterfs-fuse

# 配置 hosts

$ vi /etc/hosts

172.20.0.113   test-001.jimmysong.io
172.20.0.114   test-002.jimmysong.io
172.20.0.115   test-003.jimmysong.io
```

因为我们glusterfs跟kubernetes集群复用主机，因为此这一步可以省去。

配置 endpoints

```
$ curl -O https://raw.githubusercontent.com/kubernetes/kubernete
s/master/examples/volumes/glusterfs/glusterfs-endpoints.json

# 修改 endpoints.json ，配置 glusters 集群节点ip
# 每一个 addresses 为一个 ip 组

    {
      "addresses": [
        {
          "ip": "172.22.0.113"
        }
      ],
      "ports": [
        {
          "port": 1990
        }
      ]
    },

# 导入 glusterfs-endpoints.json

$ kubectl apply -f glusterfs-endpoints.json

# 查看 endpoints 信息
$ kubectl get ep
```

配置 service

```
$ curl -O https://raw.githubusercontent.com/kubernetes/kubernete
s/master/examples/volumes/glusterfs/glusterfs-service.json

# service.json 里面查找的是 enpointes 的名称与端口，端口默认配置为 1，
我改成了1990

# 导入 glusterfs-service.json
$ kubectl apply -f glusterfs-service.json

# 查看 service 信息
$ kubectl get svc
```

创建测试 pod

```
$ curl -O https://raw.githubusercontent.com/kubernetes/kubernete
s/master/examples/volumes/glusterfs/glusterfs-pod.json

# 编辑 glusterfs-pod.json
# 修改 volumes  下的 path 为上面创建的 volume 名称

"path": "k8s-volume"

# 导入 glusterfs-pod.json
$ kubectl apply -f glusterfs-pod.json

# 查看 pods 状态
$ kubectl get pods
NAME                             READY     STATUS    RESTARTS
AGE
glusterfs                        1/1       Running   0
1m

# 查看 pods 所在 node
$ kubectl describe pods/glusterfs

# 登陆 node 物理机，使用 df 可查看挂载目录
$ df -h
172.20.0.113:k8s-volume 1073741824        0 1073741824   0% 172.
20.0.113:k8s-volume  1.0T     0  1.0T   0% /var/lib/kubelet/pods
/3de9fc69-30b7-11e7-bfbd-8af1e3a7c5bd/volumes/kubernetes.io~glus
terfs/glusterfsvol
```

配置PersistentVolume

PersistentVolume（PV）和 PersistentVolumeClaim（PVC）是kubernetes提供的两种API资源，用于抽象存储细节。管理员关注于如何通过pv提供存储功能而无需关注用户如何使用，同样的用户只需要挂载PVC到容器中而不需要关注存储卷采用何种技术实现。

PVC和PV的关系跟pod和node关系类似，前者消耗后者的资源。PVC可以向PV申请指定大小的存储资源并设置访问模式。

PV属性

- storage容量
- 读写属性：分别为ReadWriteOnce：单个节点读写； ReadOnlyMany：多节点只读 ； ReadWriteMany：多节点读写

```
$ cat glusterfs-pv.yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: gluster-dev-volume
spec:
  capacity:
    storage: 8Gi
  accessModes:
    - ReadWriteMany
  glusterfs:
    endpoints: "glusterfs-cluster"
    path: "k8s-volume"
    readOnly: false

# 导入PV
$ kubectl apply -f glusterfs-pv.yaml

# 查看 pv
$ kubectl get pv
NAME                 CAPACITY   ACCESSMODES   RECLAIMPOLICY   ST
ATUS      CLAIM     STORAGECLASS   REASON    AGE
gluster-dev-volume   8Gi        RWX           Retain          Av
ailable                                      3s
```

PVC属性

- 访问属性与PV相同
- 容量：向PV申请的容量 <= PV总容量

配置PVC

```
$ cat glusterfs-pvc.yaml
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: glusterfs-nginx
spec:
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 8Gi

# 导入 pvc
$ kubectl apply -f glusterfs-pvc.yaml

# 查看 pvc

$ kubectl get pv
NAME              STATUS    VOLUME               CAPACITY   ACCE
SSMODES   STORAGECLASS   AGE
glusterfs-nginx   Bound     gluster-dev-volume   8Gi        RWX
                         4s
```

创建 nginx deployment 挂载 volume

```
$ vi nginx-deployment.yaml
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: nginx-dm
spec:
  replicas: 2
  template:
    metadata:
      labels:
        name: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:alpine
          imagePullPolicy: IfNotPresent
          ports:
            - containerPort: 80
          volumeMounts:
            - name: gluster-dev-volume
              mountPath: "/usr/share/nginx/html"
      volumes:
      - name: gluster-dev-volume
        persistentVolumeClaim:
```

```
          claimName: glusterfs-nginx

# 导入 deployment
$ kubectl apply -f nginx-deployment.yaml

# 查看 deployment
$ kubectl get pods |grep nginx-dm
nginx-dm-3698525684-g0mvt       1/1       Running   0          6
s
nginx-dm-3698525684-hbzq1       1/1       Running   0          6
s

# 查看 挂载
$ kubectl exec -it nginx-dm-3698525684-g0mvt -- df -h|grep k8s-v
olume
172.20.0.113:k8s-volume         1.0T     0  1.0T   0% /usr/share
/nginx/html

# 创建文件 测试
$ kubectl exec -it nginx-dm-3698525684-g0mvt -- touch /usr/share
/nginx/html/index.html

$ kubectl exec -it nginx-dm-3698525684-g0mvt -- ls -lt /usr/shar
e/nginx/html/index.html
-rw-r--r-- 1 root root 0 May  4 11:36 /usr/share/nginx/html/inde
x.html

# 验证 glusterfs
# 因为我们使用分布卷，所以可以看到某个节点中有文件
[root@test-001 ~] ls /opt/gfs_data/
[root@test-002 ~] ls /opt/gfs_data/
index.html
[root@test-003 ~] ls /opt/gfs_data/
```

参考

- CentOS 7 安装 GlusterFS
- GlusterFS with kubernetes

使用Heketi作为kubernetes的持久存储GlusterFS的external provisioner（Kubernetes集成GlusterFS集群和Heketi）

本文翻译自heketi的github网址官方文档（大部分为google翻译，少许人工调整，括号内为个人注解）其中注意事项部分为其他网上查询所得。 本文的整个过程将在kubernetes集群上的3个或以上节点安装glusterfs的服务端集群（DaemonSet方式），并将heketi以deployment的方式部署到kubernetes集群。在我的示例部分有StorageClass和PVC的样例。本文介绍的Heketi，GlusterFS这2个组件与kubernetes集成只适合用于测试验证环境，并不适合生产环境，请注意这一点。

Heketi是一个具有resetful接口的glusterfs管理程序，作为kubernetes的Storage存储的external provisioner。 “Heketi提供了一个RESTful管理界面，可用于管理GlusterFS卷的生命周期。借助Heketi，像OpenStack Manila，Kubernetes和OpenShift这样的云服务可以动态地配置GlusterFS卷和任何支持的持久性类型。Heketi将自动确定整个集群的brick位置，确保将brick及其副本放置在不同的故障域中。Heketi还支持任意数量的GlusterFS集群，允许云服务提供网络文件存储，而不受限于单个GlusterFS集群。”

注意事项

- 安装Glusterfs客户端：每个kubernetes集群的节点需要安装gulsterfs的客户端，如ubuntu系统的 `apt-get install glusterfs-client` 。
- 加载内核模块：每个kubernetes集群的节点运行 `modprobe dm_thin_pool` ，加载内核模块。
- 至少三个slave节点：至少需要3个kubernetes slave节点用来部署glusterfs集群，并且这3个slave节点每个节点需要至少一个空余的磁盘。

概述

本指南支持在Kubernetes集群中集成，部署和管理GlusterFS 容器化的存储节点。这使得Kubernetes管理员可以为其用户提供可靠的共享存储。

跟这个话题相关的另一个重要资源是gluster-kubernetes 项目。它专注于在Kubernetes集群中部署GlusterFS，并提供简化的工具来完成此任务。它包含一个安装指南 setup guide。它还包括一个样例 Hello World。其中包含一个使用动态配置（dynamically-provisioned）的GlusterFS卷进行存储的Web server pod示例。对于那些想要测试或学习更多关于此主题的人，请按照主README 的快速入门说明进行操作。

本指南旨在展示Heketi在Kubernetes环境中管理Gluster的最简单示例。这是为了强调这种配置的主要组成组件，因此并不适合生产环境。

基础设施要求

- 正在运行的Kubernetes集群，至少有三个Kubernetes工作节点，每个节点至少有一个可用的裸块设备（如EBS卷或本地磁盘）.
- 用于运行GlusterFS Pod的三个Kubernetes节点必须为GlusterFS通信打开相应的端口（如果开启了防火墙的情况下，没开防火墙就不需要这些操作）。在每个节点上运行以下命令。

```
iptables -N heketi
iptables -A heketi -p tcp -m state --state NEW -m tcp --dpor
t 24007 -j ACCEPT
iptables -A heketi -p tcp -m state --state NEW -m tcp --dpor
t 24008 -j ACCEPT
iptables -A heketi -p tcp -m state --state NEW -m tcp --dpor
t 2222 -j ACCEPT
iptables -A heketi -p tcp -m state --state NEW -m multiport
--dports 49152:49251 -j ACCEPT
service iptables save
```

客户端安装

Heketi提供了一个CLI客户端，为用户提供了一种管理Kubernetes中GlusterFS的部署和配置的方法。 在客户端机器上下载并安装Download and install the heketi-cli。

Glusterfs和Heketi在Kubernetes集群中的部署过程

以下所有文件都位于下方extras/kubernetes (`git clone https://github.com/heketi/heketi.git`)。

- 部署 GlusterFS DaemonSet

```
$ kubectl create -f glusterfs-daemonset.json
```

- 通过运行如下命令获取节点名称:

```
$ kubectl get nodes
```

- 通过设置storagenode=glusterfs节点上的标签，将gluster容器部署到指定节点上。

```
$ kubectl label node <...node...> storagenode=glusterfs
```

根据需要重复打标签的步骤。验证Pod在节点上运行至少应运行3个Pod（因此至少需要给3个节点打标签）。

```
$ kubectl get pods
```

- 接下来，我们将为Heketi创建一个服务帐户（service-account）:

```
$ kubectl create -f heketi-service-account.json
```

- 我们现在必须给该服务帐户的授权绑定相应的权限来控制gluster的pod。我们通过为我们新创建的服务帐户创建群集角色绑定（cluster role binding）来完成此操作。

```
$ kubectl create clusterrolebinding heketi-gluster-admin --clust
errole=edit --serviceaccount=default:heketi-service-account
```

- 现在我们需要创建一个Kubernetes secret来保存我们Heketi实例的配置。必须将配置文件的执行程序设置为 kubernetes才能让Heketi server控制gluster pod（配置文件的默认配置）。除此这些，可以尝试配置的其他选项。

```
$ kubectl create secret generic heketi-config-secret --from-file=
./heketi.json
```

- 接下来，我们需要部署一个初始（bootstrap）Pod和一个服务来访问该Pod。在你用git克隆的repo中，会有一个heketi-bootstrap.json文件。

提交文件并验证一切正常运行，如下所示：

```
# kubectl create -f heketi-bootstrap.json
service "deploy-heketi" created
deployment "deploy-heketi" created

# kubectl get pods
NAME                                                      READY
    STATUS    RESTARTS   AGE
deploy-heketi-1211581626-2jotm                            1/1
    Running   0          35m
glusterfs-ip-172-20-0-217.ec2.internal-1217067810-4gsvx   1/1
    Running   0          1h
glusterfs-ip-172-20-0-218.ec2.internal-2001140516-i9dw9   1/1
    Running   0          1h
glusterfs-ip-172-20-0-219.ec2.internal-2785213222-q3hba   1/1
    Running   0          1h
```

- 当Bootstrap heketi服务正在运行，我们配置端口转发，以便我们可以使用Heketi CLI与服务进行通信。使用heketi pod的名称，运行下面的命令：

```
kubectl port-forward deploy-heketi-1211581626-2jotm :8080
```

如果在运行命令的系统上本地端口8080是空闲的，则可以运行port-forward命令，以便绑定到8080以方便使用（2个命令二选一即可，我选择第二个）：

```
kubectl port-forward deploy-heketi-1211581626-2jotm 8080:8080
```

现在通过对Heketi服务运行示例查询来验证端口转发是否正常。该命令应该已经打印了将从其转发的本地端口。将其合并到URL中以测试服务，如下所示：

```
curl http://localhost:8080/hello
Handling connection for 8080
Hello from heketi
```

最后，为Heketi CLI客户端设置一个环境变量，以便它知道Heketi服务器的地址。

```
export HEKETI_CLI_SERVER=http://localhost:8080
```

- 接下来，我们将向Heketi提供有关要管理的GlusterFS集群的信息。通过拓扑文件提供这些信息。克隆的repo中有一个示例拓扑文件，名为topology-sample.json。拓扑指定运行GlusterFS容器的Kubernetes节点以及每个节点的相应原始块设备。

确保hostnames/manage指向如下所示的确切名称kubectl get nodes得到的主机名（如ubuntu-1），并且hostnames/storage是存储网络的IP地址（对应ubuntu-1的ip地址）。

IMPORTANT: 重要提示，目前，必须使用与服务器版本匹配的Heketi-cli版本加载拓扑文件。另外，Heketi pod 带有可以通过 `kubectl exec ...` 访问的heketi-cli副本。

修改拓扑文件以反映您所做的选择，然后如下所示部署它（修改主机名，IP，block设备的名称 如xvdg）：

```
heketi-client/bin/heketi-cli topology load --json=topology-sampl
e.json
Handling connection for 57598
    Found node ip-172-20-0-217.ec2.internal on cluster e6c063ba3
98f8e9c88a6ed720dc07dd2
        Adding device /dev/xvdg ... OK
    Found node ip-172-20-0-218.ec2.internal on cluster e6c063ba3
98f8e9c88a6ed720dc07dd2
        Adding device /dev/xvdg ... OK
    Found node ip-172-20-0-219.ec2.internal on cluster e6c063ba3
98f8e9c88a6ed720dc07dd2
        Adding device /dev/xvdg ... OK
```

- 接下来，我们将使用heketi为其存储其数据库提供一个卷（不要怀疑，就是使用这个命令，openshift和kubernetes通用，此命令生成heketi-storage.json文件）：

```
# heketi-client/bin/heketi-cli setup-openshift-heketi-storage
# kubectl create -f heketi-storage.json
```

> Pitfall: 注意，如果在运行setup-openshift-heketi-storage子命令时heketi-cli报告“无空间”错误，则可能无意中运行topology load命令的时候服务端和heketi-cli的版本不匹配造成的。停止正在运行的heketi pod（kubectl scale deployment deploy-heketi --replicas=0），手动删除存储块设备中的任何签名，然后继续运行heketi pod（kubectl scale deployment deploy-heketi --replicas=1）。然后用匹配版本的heketi-cli重新加载拓扑，然后重试该步骤。

- 等到作业完成后，删除bootstrap Heketi实例相关的组件：

```
# kubectl delete all,service,jobs,deployment,secret --selector="
deploy-heketi"
```

- 创建长期使用的Heketi实例（存储持久化的）：

```
# kubectl create -f heketi-deployment.json
service "heketi" created
deployment "heketi" created
```

- 这样做了以后，heketi db将使用GlusterFS卷，并且每当heketi pod重新启动时都不会重置（数据不会丢失，存储持久化）。

使用诸如heketi-cli cluster list和的命令heketi-cli volume list 来确认先前建立的集群存在，并且heketi可以列出在bootstrap阶段创建的db存储卷。

使用样例

有两种方法来调配存储。常用的方法是设置一个StorageClass，让Kubernetes为提交的PersistentVolumeClaim自动配置存储。或者，可以通过Kubernetes手动创建和管理卷（PVs），或直接使用heketi-cli中的卷。

参考gluster-kubernetes hello world example 获取关于 storageClass 的更多信息.

我的示例（非翻译部分内容）

- topology文件：我的例子（3个节点，ubuntu-1（192.168.5.191）,ubuntu-2（192.168.5.192）,ubuntu-3（192.168.5.193）,每个节点2个磁盘用来做存储（sdb，sdc））

```
# cat topology-sample.json


{
  "clusters": [
    {
      "nodes": [
        {
          "node": {
            "hostnames": {
              "manage": [
                "ubuntu-1"
              ],
              "storage": [
                "192.168.5.191"
              ]
            },
            "zone": 1
          },
          "devices": [
            "/dev/sdb",
            "/dev/sdc"
          ]
        },
        {
          "node": {
            "hostnames": {
              "manage": [
                "ubuntu-2"
              ],
              "storage": [
                "192.168.5.192"
              ]
            },
            "zone": 1
          },
          "devices": [
            "/dev/sdb",
            "/dev/sdc"
          ]
        },
        {
          "node": {
            "hostnames": {
              "manage": [
                "ubuntu-3"
              ],
              "storage": [
                "192.168.5.193"
              ]
            },
```

```
                "zone": 1
              },
              "devices": [
                "/dev/sdb",
                "/dev/sdc"
              ]
            }
          ]
        }
      ]
    }
```

- 确认glusterfs和heketi的pod运行正常

```
# kubectl get pod
NAME                                                         READY
    STATUS     RESTARTS   AGE
glusterfs-gf5zc                                              1/1
    Running    2          8h
glusterfs-ngc55                                              1/1
    Running    2          8h
glusterfs-zncjs                                              1/1
    Running    0          2h
heketi-5c8ffcc756-x9gnv                                      1/1
    Running    5          7h
```

- StorageClass yaml文件示例

```
# cat storage-class-slow.yaml


apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: slow                                           #-------------SC
的名字
provisioner: kubernetes.io/glusterfs
parameters:
  resturl: "http://10.103.98.75:8080"                  #-------------hek
eti service的cluster ip 和端口
  restuser: "admin"                                    #-------------随便
填，因为没有启用鉴权模式
  gidMin: "40000"
  gidMax: "50000"
  volumetype: "replicate:3"                            #-------------申请
的默认为3副本模式
```

- PVC举例

```
# cat pvc-sample.yaml


kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: myclaim
  annotations:
    volume.beta.kubernetes.io/storage-class: "slow"           #-
------------sc的名字,需要与storageclass的名字一致
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
```

查看创建的pvc和pv

```
# kubectl get pvc|grep myclaim
NAME                                 STATUS     VOLUME
                 CAPACITY   ACCESS MODES   STORAGECLASS   AGE
myclaim                              Bound      pvc-e98e9117-3ed7-11e8-b61
d-08002795cb26   1Gi        RWO            slow           28s

# kubectl get pv|grep myclaim
NAME                                       CAPACITY   ACCESS MOD
ES    RECLAIM POLICY   STATUS        CLAIM
         STORAGECLASS   REASON     AGE
pvc-e98e9117-3ed7-11e8-b61d-08002795cb26   1Gi        RWO
     Delete            Bound      default/myclaim
    slow                         1m
```

- 可以将slow的sc设置为默认，这样平台分配存储的时候可以自动从glusterfs集群分配pv

```
# kubectl patch storageclass slow -p '{"metadata": {"annotations
":{"storageclass.kubernetes.io/is-default-class":"true"}}}'
storageclass.storage.k8s.io "slow" patched

# kubectl get sc
NAME             PROVISIONER                AGE
default          fuseim.pri/ifs             1d
slow (default)   kubernetes.io/glusterfs    6h
```

容量限额测试

已经通过Helm 部署的一个mysql2 实例，使用存储2G，信息查看如下：

```
# helm list
NAME        REVISION    UPDATED                     STATUS
CHART                   NAMESPACE
mysql2      1           Thu Apr 12 15:27:11 2018    DEPLOYED
mysql-0.3.7             default
```

查看PVC和PV，大小2G，mysql2-mysql

```
# kubectl get pvc
NAME                        STATUS    VOLUME
                CAPACITY    ACCESS MODES   STORAGECLASS   AGE
mysql2-mysql                Bound     pvc-ea4ae3e0-3e22-11e8-8bb
6-08002795cb26   2Gi        RWO            slow           19h

# kubectl get pv
NAME                                          CAPACITY   ACCESS MOD
ES    RECLAIM POLICY   STATUS        CLAIM
        STORAGECLASS   REASON    AGE
pvc-ea4ae3e0-3e22-11e8-8bb6-08002795cb26   2Gi        RWO
     Delete           Bound         default/mysql2-mysql
        slow                     19h
```

查看mysql的pod

```
# kubectl get pod|grep mysql2
mysql2-mysql-56d64f5b77-j2v84                             1/1
    Running   2          19h
```

进入mysql所在容器

```
# kubectl exec -it mysql2-mysql-56d64f5b77-j2v84  /bin/bash
```

查看挂载路径，查看挂载信息

```
root@mysql2-mysql-56d64f5b77-j2v84:/#cd /var/lib/mysql
root@mysql2-mysql-56d64f5b77-j2v84:/var/lib/mysql#
root@mysql2-mysql-56d64f5b77-j2v84:/var/lib/mysql# df -h
Filesystem                                          Size  Used A
vail Use% Mounted on
none                                                 48G  9.2G
 37G  21% /
tmpfs                                               1.5G     0
1.5G   0% /dev
tmpfs                                               1.5G     0
1.5G   0% /sys/fs/cgroup
/dev/mapper/ubuntu--1--vg-root                       48G  9.2G
 37G  21% /etc/hosts
shm                                                  64M     0
 64M   0% /dev/shm
192.168.5.191:vol_2c2227ee65b64a0225aa9bce848a9925  2.0G  264M
1.8G  13% /var/lib/mysql
tmpfs                                               1.5G   12K
1.5G   1% /run/secrets/kubernetes.io/serviceaccount
tmpfs                                               1.5G     0
1.5G   0% /sys/firmware
```

使用dd写入数据，写入一段时间以后，空间满了，会报错（报错信息有bug，不是报空间满了，而是报文件系统只读，应该是glusterfs和docker配合的问题）

```
root@mysql2-mysql-56d64f5b77-j2v84:/var/lib/mysql# dd if=/dev/ze
ro of=test.img bs=8M count=300

dd: error writing 'test.img': Read-only file system
dd: closing output file 'test.img': Input/output error
```

查看写满以后的文件大小

```
root@mysql2-mysql-56d64f5b77-j2v84:/var/lib/mysql# ls -l
total 2024662
-rw-r----- 1 mysql mysql         56 Apr 12 07:27 auto.cnf
-rw-r----- 1 mysql mysql       1329 Apr 12 07:27 ib_buffer_pool
-rw-r----- 1 mysql mysql   50331648 Apr 12 12:05 ib_logfile0
-rw-r----- 1 mysql mysql   50331648 Apr 12 07:27 ib_logfile1
-rw-r----- 1 mysql mysql   79691776 Apr 12 12:05 ibdata1
-rw-r----- 1 mysql mysql   12582912 Apr 12 12:05 ibtmp1
drwxr-s--- 2 mysql mysql       8192 Apr 12 07:27 mysql
drwxr-s--- 2 mysql mysql       8192 Apr 12 07:27 performance_sch
ema
drwxr-s--- 2 mysql mysql       8192 Apr 12 07:27 sys
-rw-r--r-- 1 root  mysql 1880887296 Apr 13 02:47 test.img
```

查看挂载信息（挂载信息显示bug，应该是glusterfs的bug）

```
root@mysql2-mysql-56d64f5b77-j2v84:/var/lib/mysql# df -h
Filesystem                                         Size  Used A
vail Use% Mounted on
none                                                48G  9.2G
 37G  21% /
tmpfs                                              1.5G     0
1.5G   0% /dev
tmpfs                                              1.5G     0
1.5G   0% /sys/fs/cgroup
/dev/mapper/ubuntu--1--vg-root                      48G  9.2G
 37G  21% /etc/hosts
shm                                                 64M     0
 64M   0% /dev/shm
192.168.5.191:vol_2c2227ee65b64a0225aa9bce848a9925  2.0G  -16E
   0 100% /var/lib/mysql
tmpfs                                              1.5G   12K
1.5G   1% /run/secrets/kubernetes.io/serviceaccount
tmpfs                                              1.5G     0
1.5G   0% /sys/firmware
```

查看文件夹大小，为2G

```
# du -h
25M         ./mysql
825K    ./performance_schema
496K    ./sys
2.0G    .
```

如上说明glusterfs的限额作用是起效的，限制在2G的空间大小。

在OpenShift中使用GlusterFS做持久化存储

概述

本文由Daniel Messer（Technical Marketing Manager Storage @RedHat）和Keith Tenzer（Solutions Architect @RedHat）共同撰写。

- Storage for Containers Overview – Part I
- Storage for Containers using Gluster – Part II
- Storage for Containers using Container Native Storage – Part III
- Storage for Containers using Ceph – Part IV
- Storage for Containers using NetApp ONTAP NAS – Part V
- Storage for Containers using NetApp SolidFire – Part VI

Gluster作为Container-Ready Storage(CRS)

在本文中，我们将介绍容器存储的首选以及如何部署它。 Kusternet和OpenShift支持GlusterFS已经有一段时间了。 GlusterFS的适用性很好，可用于所有的部署场景：裸机、虚拟机、内部部署和公共云。 在容器中运行GlusterFS的新特性将在本系列后面讨论。

GlusterFS是一个分布式文件系统，内置了原生协议（GlusterFS）和各种其他协议（NFS，SMB，...）。 为了与OpenShift集成，节点将通过FUSE使用原生协议，将GlusterFS卷挂在到节点本身上，然后将它们绑定到目标容器中。 OpenShift / Kubernetes具有实现请求、释放和挂载、卸载GlusterFS卷的原生程序。

CRS概述

在存储方面，根据OpenShift / Kubernetes的要求，还有一个额外的组件管理集群，称为“heketi”。 这实际上是一个用于GlusterFS的REST API，它还提供CLI版本。 在以下步骤中，我们将在3个GlusterFS节点中部署heketi，使用它来部署GlusterFS存储池，将其连接到OpenShift，并使用它来通过PersistentVolumeClaims为容器配置存储。 我们将总共部署4台虚拟机。 一个用于OpenShift（实验室设置），另一个用于GlusterFS。

注意：您的系统应至少需要有四核CPU，16GB RAM和20 GB可用磁盘空间。

部署OpenShift

首先你需要先部署OpenShift。最有效率的方式是直接在虚拟机中部署一个All-in-One环境，部署指南见 the “OpenShift Enterprise 3.4 all-in-one Lab Environment” article.。

确保你的OpenShift虚拟机可以解析外部域名。编辑 `/etc/dnsmasq.conf` 文件，增加下面的Google DNS：

```
server=8.8.8.8
```

重启：

```
# systemctl restart dnsmasq
# ping -c1 google.com
```

部署Gluster

GlusterFS至少需要有以下配置的3台虚拟机：

- RHEL 7.3
- 2 CPUs
- 2 GB内存
- 30 GB磁盘存储给操作系统
- 10 GB磁盘存储给GlusterFS bricks

修改/etc/hosts文件，定义三台虚拟机的主机名。

例如（主机名可以根据你自己的环境自由调整）

```
# cat /etc/hosts
127.0.0.1      localhost localhost.localdomain localhost4 localh
ost4.localdomain4
::1            localhost localhost.localdomain localhost6 localh
ost6.localdomain6
172.16.99.144  ocp-master.lab ocp-master
172.16.128.7   crs-node1.lab crs-node1
172.16.128.8   crs-node2.lab crs-node2
172.16.128.9   crs-node3.lab crs-node3
```

在3台GlusterFS虚拟机上都执行以下步骤：

```
# subscription-manager repos --disable="*"
# subscription-manager repos --enable=rhel-7-server-rpms
```

如果你已经订阅了GlusterFS那么可以直接使用，开启 `rh-gluster-3-for-rhel-7-server-rpms` 的yum源。

如果你没有的话，那么可以通过EPEL使用非官方支持的GlusterFS的社区源。

```
# yum -y install http://dl.fedoraproject.org/pub/epel/epel-relea
se-latest-7.noarch.rpm
# rpm --import http://dl.fedoraproject.org/pub/epel/RPM-GPG-KEY-
EPEL-7
```

在 `/etc/yum.repos.d/` 目录下创建 `glusterfs-3.10.repo` 文件：

```
[glusterfs-3.10]
name=glusterfs-3.10
description="GlusterFS 3.10 Community Version"
baseurl=https://buildlogs.centos.org/centos/7/storage/x86_64/glu
ster-3.10/
gpgcheck=0
enabled=1
```

验证源已经被激活。

```
# yum repolist
```

现在可以开始安装GlusterFS了。

```
# yum -y install glusterfs-server
```

需要为GlusterFS peers打开几个基本TCP端口，以便与OpenShift进行通信并提供存储：

```
# firewall-cmd --add-port=24007-24008/tcp --add-port=49152-49664
/tcp --add-port=2222/tcp
# firewall-cmd --runtime-to-permanent
```

现在我们可以启动GlusterFS的daemon进程了：

```
# systemctl enable glusterd
# systemctl start glusterd
```

完成。GlusterFS已经启动并正在运行。其他配置将通过heketi完成。

在GlusterFS的一台虚拟机上安装heketi

```
[root@crs-node1 ~]# yum -y install heketi heketi-client
```

更新EPEL

如果你没有Red Hat Gluster Storage订阅的话，你可以从EPEL中获取heketi。 在撰写本文时，2016年10月那时候还是3.0.0-1.el7版本，它不适用于OpenShift 3.4。你将需要更新到更新的版本：

```
[root@crs-node1 ~]# yum -y install wget
[root@crs-node1 ~]# wget https://github.com/heketi/heketi/releas
es/download/v4.0.0/heketi-v4.0.0.linux.amd64.tar.gz
[root@crs-node1 ~]# tar -xzf heketi-v4.0.0.linux.amd64.tar.gz
[root@crs-node1 ~]# systemctl stop heketi
[root@crs-node1 ~]# cp heketi/heketi* /usr/bin/
[root@crs-node1 ~]# chown heketi:heketi /usr/bin/heketi*
```

在 `/etc/systemd/system/heketi.service` 中创建v4版本的heketi二进制文件的更新语法文件：

```
[Unit]
Description=Heketi Server

[Service]
Type=simple
WorkingDirectory=/var/lib/heketi
EnvironmentFile=-/etc/heketi/heketi.json
User=heketi
ExecStart=/usr/bin/heketi --config=/etc/heketi/heketi.json
Restart=on-failure
StandardOutput=syslog
StandardError=syslog

[Install]
WantedBy=multi-user.target
```

```
[root@crs-node1 ~]# systemctl daemon-reload
[root@crs-node1 ~]# systemctl start heketi
```

Heketi使用SSH来配置GlusterFS的所有节点。创建SSH密钥对，将公钥拷贝到所有3个节点上（包括你登陆的第一个节点）：

```
[root@crs-node1 ~]# ssh-keygen -f /etc/heketi/heketi_key -t rsa
-N ''
[root@crs-node1 ~]# ssh-copy-id -i /etc/heketi/heketi_key.pub ro
ot@crs-node1.lab
[root@crs-node1 ~]# ssh-copy-id -i /etc/heketi/heketi_key.pub ro
ot@crs-node2.lab
[root@crs-node1 ~]# ssh-copy-id -i /etc/heketi/heketi_key.pub ro
ot@crs-node3.lab
[root@crs-node1 ~]# chown heketi:heketi /etc/heketi/heketi_key*
```

剩下唯一要做的事情就是配置heketi来使用SSH。编辑 `/etc/heketi/heketi.json` 文件使它看起来像下面这个样子（改变的部分突出显示下划线）：

```
{
   "_port_comment":"Heketi Server Port Number",
   "port":"8080",
   "_use_auth":"Enable JWT authorization. Please enable for depl
oyment",
   "use_auth":false,
   "_jwt":"Private keys for access",
   "jwt":{
      "_admin":"Admin has access to all APIs",
```

```
         "admin":{
            "key":"My Secret"
         },
         "_user":"User only has access to /volumes endpoint",
         "user":{
            "key":"My Secret"
         }
      },
      "_glusterfs_comment":"GlusterFS Configuration",
      "glusterfs":{
         "_executor_comment":[
            "Execute plugin. Possible choices: mock, ssh",
            "mock: This setting is used for testing and development
.",
            " It will not send commands to any node.",
            "ssh: This setting will notify Heketi to ssh to the nod
es.",
            " It will need the values in sshexec to be configured.",

            "kubernetes: Communicate with GlusterFS containers over"
,
            " Kubernetes exec api."
         ],
         "executor":"ssh",
         "_sshexec_comment":"SSH username and private key file info
rmation",
         "sshexec":{
            "keyfile":"/etc/heketi/heketi_key",
            "user":"root",
            "port":"22",
            "fstab":"/etc/fstab"
         },
         "_kubeexec_comment":"Kubernetes configuration",
         "kubeexec":{
            "host":"https://kubernetes.host:8443",
            "cert":"/path/to/crt.file",
            "insecure":false,
            "user":"kubernetes username",
            "password":"password for kubernetes user",
            "namespace":"OpenShift project or Kubernetes namespace",

            "fstab":"Optional: Specify fstab file on node. Default
is /etc/fstab"
         },
         "_db_comment":"Database file name",
         "db":"/var/lib/heketi/heketi.db",
         "_loglevel_comment":[
            "Set log level. Choices are:",
            " none, critical, error, warning, info, debug",
            "Default is warning"
         ],
         "loglevel":"debug"
      }
```

```
}
```

完成。heketi将监听8080端口，我们来确认下防火墙规则允许它监听该端口：

```
# firewall-cmd --add-port=8080/tcp
# firewall-cmd --runtime-to-permanent
```

重启heketi：

```
# systemctl enable heketi
# systemctl restart heketi
```

测试它是否在运行：

```
# curl http://crs-node1.lab:8080/hello
Hello from Heketi
```

很好。heketi上场的时候到了。 我们将使用它来配置我们的GlusterFS存储池。 该软件已经在我们所有的虚拟机上运行，但并未被配置。 要将其改造为满足我们需求的存储系统，需要在拓扑文件中描述我们所需的GlusterFS存储池，如下所示：

```
# vi topology.json
{
  "clusters": [
    {
      "nodes": [
        {
          "node": {
            "hostnames": {
              "manage": [
                "crs-node1.lab"
              ],
              "storage": [
                "172.16.128.7"
              ]
            },
            "zone": 1
          },
          "devices": [
            "/dev/sdb"
          ]
        },
        {
```

```
          "node": {
            "hostnames": {
              "manage": [
                "crs-node2.lab"
              ],
              "storage": [
                "172.16.128.8"
              ]
            },
            "zone": 1
          },
          "devices": [
            "/dev/sdb"
          ]
        },
        {
          "node": {
            "hostnames": {
              "manage": [
                "crs-node3.lab"
              ],
              "storage": [
                "172.16.128.9"
              ]
            },
            "zone": 1
          },
          "devices": [
            "/dev/sdb"
          ]
        }
      ]
    }
  ]
}
```

该文件格式比较简单，基本上是告诉heketi要创建一个3节点的集群，其中每个节点包含的配置有FQDN，IP地址以及至少一个将用作GlusterFS块的备用块设备。

现在将该文件发送给heketi：

```
# export HEKETI_CLI_SERVER=http://crs-node1.lab:8080
# heketi-cli topology load --json=topology.json
Creating cluster ... ID: 78cdb57aa362f5284bc95b2549bc7e7d
 Creating node crs-node1.lab ... ID: ffd7671c0083d88aeda9fd1cb40
b339b
 Adding device /dev/sdb ... OK
 Creating node crs-node2.lab ... ID: 8220975c0a4479792e684584153
050a9
 Adding device /dev/sdb ... OK
 Creating node crs-node3.lab ... ID: b94f14c4dbd8850f6ac589ac3b3
9cc8e
 Adding device /dev/sdb ... OK
```

现在heketi已经配置了3个节点的GlusterFS存储池。很简单！你现在可以看到3个虚拟机都已经成功构成了GlusterFS中的可信存储池（Trusted Stroage Pool）。

```
[root@crs-node1 ~]# gluster peer status
Number of Peers: 2

Hostname: crs-node2.lab
Uuid: 93b34946-9571-46a8-983c-c9f128557c0e
State: Peer in Cluster (Connected)
Other names:
crs-node2.lab

Hostname: 172.16.128.9
Uuid: e3c1f9b0-be97-42e5-beda-f70fc05f47ea
State: Peer in Cluster (Connected)
```

现在回到OpenShift！

将Gluster与OpenShift集成

为了集成OpenShift，需要两样东西：一个动态的Kubernetes Storage Provisioner和一个StorageClass。 Provisioner在OpenShift中开箱即用。 实际上关键的是如何将存储挂载到容器上。 StorageClass是OpenShift中的用户可以用来实现的PersistentVolumeClaims的实体，它反过来能够触发一个Provisioner实现实际的配置，并将结果表示为Kubernetes PersistentVolume（PV）。

就像OpenShift中的其他组件一样，StorageClass也简单的用YAML文件定义：

```
# cat crs-storageclass.yaml
kind: StorageClass
apiVersion: storage.k8s.io/v1beta1
metadata:
 name: container-ready-storage
 annotations:
 storageclass.beta.kubernetes.io/is-default-class: "true"
provisioner: kubernetes.io/glusterfs
parameters:
 resturl: "http://crs-node1.lab:8080"
 restauthenabled: "false"
```

我们的provisioner是kubernetes.io/glusterfs，将它指向我们的heketi实例。 我们将类命名为“container-ready-storage”，同时使其成为所有没有显示指定StorageClass的PersistentVolumeClaim的默认StorageClass。

为你的GlusterFS池创建StorageClass：

```
# oc create -f crs-storageclass.yaml
```

在OpenShift中使用Gluster

我们来看下如何在OpenShift中使用GlusterFS。首先在OpenShift虚拟机中创建一个测试项目。

```
# oc new-project crs-storage --display-name="Container-Ready Sto
rage"
```

这会向Kubernetes/OpenShift发出storage请求，请求一个PersistentVolumeClaim（PVC）。 这是一个简单的对象，它描述最少需要多少容量和应该提供哪种访问模式（非共享，共享，只读）。 它通常是应用程序模板的一部分，但我们只需创建一个独立的PVC：

```
# cat crs-claim.yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
 name: my-crs-storage
 namespace: crs-storage
spec:
 accessModes:
 - ReadWriteOnce
 resources:
 requests:
 storage: 1Gi
```

发送该请求：

```
# oc create -f crs-claim.yaml
```

观察在OpenShfit中，PVC正在以动态创建volume的方式实现：

```
# oc get pvc
NAME             STATUS    VOLUME
      CAPACITY   ACCESSMODES   AGE
my-crs-storage   Bound     pvc-41ad5adb-107c-11e7-afae-000c2949c
ce7   1Gi        RWO           58s
```

太棒了！ 你现在可以在OpenShift中使用存储容量，而不需要直接与存储系统进行任何交互。 我们来看看创建的volume：

```
# oc get pv/pvc-41ad5adb-107c-11e7-afae-000c2949cce7
Name:          pvc-41ad5adb-107c-11e7-afae-000c2949cce7
Labels:
StorageClass:     container-ready-storage
Status:         Bound
Claim:          crs-storage/my-crs-storage
Reclaim Policy:     Delete
Access Modes:     RWO
Capacity:     1Gi
Message:
Source:
    Type:         Glusterfs (a Glusterfs mount on the host that s
hares a pod's lifetime)
    EndpointsName:     gluster-dynamic-my-crs-storage
    Path:         vol_85e444ee3bc154de084976a9aef16025
    ReadOnly:         false
```

What happened in the background was that when the PVC reached the system, our default StorageClass reached out to the GlusterFS Provisioner with the volume specs from the PVC. The provisioner in turn communicates with our heketi instance which facilitates the creation of the GlusterFS volume, which we can trace in it's log messages:

该volume是根据PVC中的定义特别创建的。 在PVC中，我们没有明确指定要使用哪个StorageClass，因为heketi的GlusterFS StorageClass已经被定义为系统范围的默认值。

在后台发生的情况是，当PVC到达系统时，默认的StorageClass请求具有该PVC中volume声明规格的GlusterFS Provisioner。 Provisioner又与我们的heketi实例通信，这有助于创建GlusterFS volume，我们可以在其日志消息中追踪：

```
[root@crs-node1 ~]# journalctl -l -u heketi.service
...
Mar 24 11:25:52 crs-node1.lab heketi[2598]: [heketi] DEBUG 2017/
03/24 11:25:52 /src/github.com/heketi/heketi/apps/glusterfs/volu
me_entry.go:298: Volume to be created on cluster e
Mar 24 11:25:52 crs-node1.lab heketi[2598]: [heketi] INFO 2017/0
3/24 11:25:52 Creating brick 9e791b1daa12af783c9195941fe63103
Mar 24 11:25:52 crs-node1.lab heketi[2598]: [heketi] INFO 2017/0
3/24 11:25:52 Creating brick 3e06af2f855bef521a95ada91680d14b
Mar 24 11:25:52 crs-node1.lab heketi[2598]: [heketi] INFO 2017/0
3/24 11:25:52 Creating brick e4daa240f1359071e3f7ea22618cfbab
...
Mar 24 11:25:52 crs-node1.lab heketi[2598]: [sshexec] INFO 2017/
03/24 11:25:52 Creating volume vol_85e444ee3bc154de084976a9aef16
025 replica 3
...
Mar 24 11:25:53 crs-node1.lab heketi[2598]: Result: volume creat
e: vol_85e444ee3bc154de084976a9aef16025: success: please start t
he volume to access data
...
Mar 24 11:25:55 crs-node1.lab heketi[2598]: Result: volume start
: vol_85e444ee3bc154de084976a9aef16025: success
...
Mar 24 11:25:55 crs-node1.lab heketi[2598]: [asynchttp] INFO 201
7/03/24 11:25:55 Completed job c3d6c4f9fc74796f4a5262647dc790fe
in 3.176522702s
...
```

成功！ 大约用了3秒钟，GlusterFS池就配置完成了，并配置了一个volume。 默认值是replica 3，这意味着数据将被复制到3个不同节点的3个块上（用GlusterFS作为后端存储）。 该过程是通过Heketi在OpenShift进行编排的。

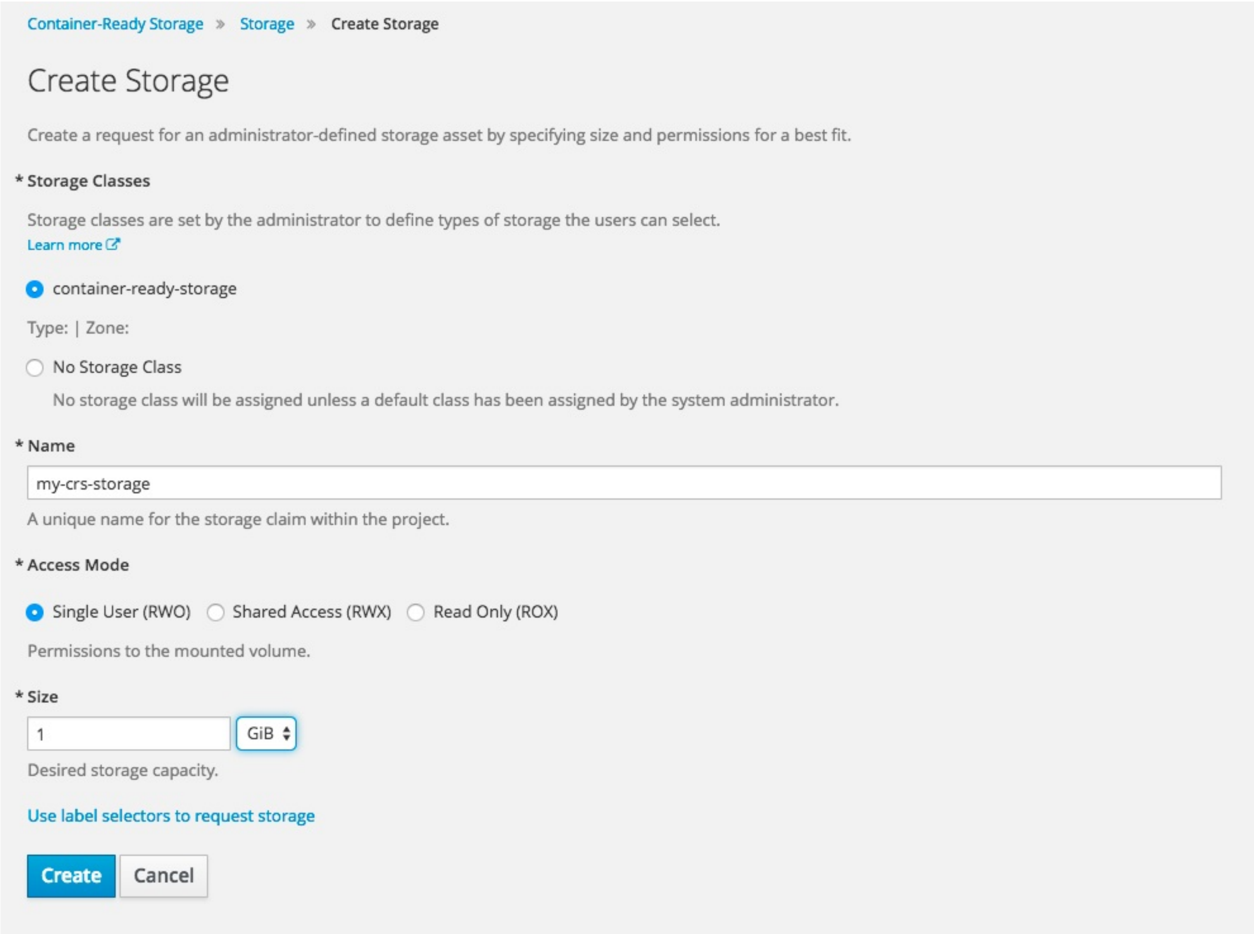
你也可以从GlusterFS的角度看到有关volume的信息：

```
[root@crs-node1 ~]# gluster volume list
vol_85e444ee3bc154de084976a9aef16025
[root@crs-node1 ~]# gluster volume info vol_85e444ee3bc154de0849
76a9aef16025

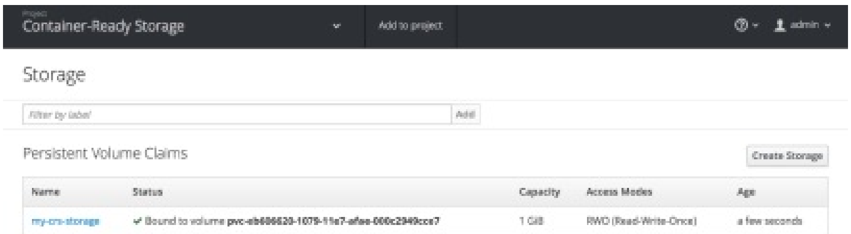
Volume Name: vol_85e444ee3bc154de084976a9aef16025
Type: Replicate
Volume ID: a32168c8-858e-472a-b145-08c20192082b
Status: Started
Snapshot Count: 0
Number of Bricks: 1 x 3 = 3
Transport-type: tcp
Bricks:
Brick1: 172.16.128.8:/var/lib/heketi/mounts/vg_147b43f6f6903be8b
23209903b7172ae/brick_9e791b1daa12af783c9195941fe63103/brick
Brick2: 172.16.128.9:/var/lib/heketi/mounts/vg_72c0f520b0c57d807
be21e9c90312f85/brick_3e06af2f855bef521a95ada91680d14b/brick
Brick3: 172.16.128.7:/var/lib/heketi/mounts/vg_67314f879686de975
f9b8936ae43c5c5/brick_e4daa240f1359071e3f7ea22618cfbab/brick
Options Reconfigured:
transport.address-family: inet
nfs.disable: on
```

请注意，GlusterFS中的卷名称如何对应于OpenShift中Kubernetes Persistent Volume的“路径”。

或者，你也可以使用OpenShift UI来配置存储，这样可以很方便地在系统中的所有已知的StorageClasses中进行选择：



图片 - 创建存储



图片 - *Screen Shot 2017-03-24 at 11.09.34.png*

让我们做点更有趣的事情，在OpenShift中运行工作负载。

在仍运行着crs-storage项目的OpenShift虚拟机中执行：

```
# oc get templates -n openshift
```

你应该可以看到一个应用程序和数据库模板列表，这个列表将方便你更轻松的使用OpenShift来部署你的应用程序项目。

我们将使用MySQL来演示如何在OpenShift上部署具有持久化和弹性存储的有状态应用程序。 Mysql-persistent模板包含一个用于MySQL数据库目录的1G空间的PVC。 为了演示目的，可以直接使用默认值。

```
# oc process mysql-persistent -n openshift | oc create -f -
```

等待部署完成。你可以通过UI或者命令行观察部署进度：

```
# oc get pods
NAME            READY     STATUS    RESTARTS   AGE
mysql-1-h4afb   1/1       Running   0          2m
```

好了。我们已经使用这个模板创建了一个service，secrets、PVC和pod。我们来使用它（你的pod名字将跟我的不同）：

```
# oc rsh mysql-1-h4afb
```

你已经成功的将它挂载到MySQL的pod上。我们连接一下数据库试试：

```
sh-4.2$ mysql -u $MYSQL_USER -p$MYSQL_PASSWORD -h $HOSTNAME $MYS
QL_DATABASE
```

这点很方便，所有重要的配置，如MySQL凭据，数据库名称等都是pod模板中的环境变量的一部分，因此可以在pod中作为shell的环境变量。 我们来创建一些数据：

```
mysql> show databases;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| sampledb           |
+--------------------+
2 rows in set (0.02 sec)

mysql> \u sampledb
Database changed
mysql> CREATE TABLE IF NOT EXISTS equipment (
    ->     equip_id int(5) NOT NULL AUTO_INCREMENT,
    ->     type varchar(50) DEFAULT NULL,
    ->     install_date DATE DEFAULT NULL,
    ->     color varchar(20) DEFAULT NULL,
    ->     working bool DEFAULT NULL,
    ->     location varchar(250) DEFAULT NULL,
    ->     PRIMARY KEY(equip_id)
    ->     );
Query OK, 0 rows affected (0.13 sec)

mysql> INSERT INTO equipment (type, install_date, color, working
, location)
    -> VALUES
    -> ("Slide", Now(), "blue", 1, "Southwest Corner");
Query OK, 1 row affected, 1 warning (0.01 sec)

mysql> SELECT * FROM equipment;
+----------+-------+--------------+-------+---------+-----------
-------+
| equip_id | type  | install_date | color | working | location
       |
+----------+-------+--------------+-------+---------+-----------
-------+
|        1 | Slide | 2017-03-24   | blue  |       1 | Southwest
Corner |
+----------+-------+--------------+-------+---------+-----------
-------+
1 row in set (0.00 sec)
```

很好，数据库运行正常。

你想看下数据存储在哪里吗？很简单！查看刚使用模板创建的mysql volume：

```
# oc get pvc/mysql
NAME      STATUS    VOLUME                                     C
APACITY   ACCESSMODES   AGE
mysql     Bound     pvc-a678b583-1082-11e7-afae-000c2949cce7   1
Gi        RWO           11m
# oc describe pv/pvc-a678b583-1082-11e7-afae-000c2949cce7
Name:           pvc-a678b583-1082-11e7-afae-000c2949cce7
Labels:
StorageClass:    container-ready-storage
Status:          Bound
Claim:          crs-storage/mysql
Reclaim Policy:    Delete
Access Modes:    RWO
Capacity:    1Gi
Message:
Source:
    Type:        Glusterfs (a Glusterfs mount on the host that s
hares a pod's lifetime)
    EndpointsName:    gluster-dynamic-mysql
    Path:        vol_6299fc74eee513119dafd43f8a438db1
    ReadOnly:        false
```

GlusterFS的volume名字是vol_6299fc74eee513119dafd43f8a438db1。回到你的GlusterFS虚拟机中，输入：

```
# gluster volume info vol_6299fc74eee513119dafd43f8a438db

Volume Name: vol_6299fc74eee513119dafd43f8a438db1
Type: Replicate
Volume ID: 4115918f-28f7-4d4a-b3f5-4b9afe5b391f
Status: Started
Snapshot Count: 0
Number of Bricks: 1 x 3 = 3
Transport-type: tcp
Bricks:
Brick1: 172.16.128.7:/var/lib/heketi/mounts/vg_67314f879686de975
f9b8936ae43c5c5/brick_f264a47aa32be5d595f83477572becf8/brick
Brick2: 172.16.128.8:/var/lib/heketi/mounts/vg_147b43f6f6903be8b
23209903b7172ae/brick_f5731fe7175cbe6e6567e013c2591343/brick
Brick3: 172.16.128.9:/var/lib/heketi/mounts/vg_72c0f520b0c57d807
be21e9c90312f85/brick_ac6add804a6a467cd81cd1404841bbf1/brick
Options Reconfigured:
transport.address-family: inet
nfs.disable: on
```

你可以看到数据是如何被复制到3个GlusterFS块的。我们从中挑一个（最好挑选你刚登陆的那台虚拟机并查看目录）：

```
# ll /var/lib/heketi/mounts/vg_67314f879686de975f9b8936ae43c5c5/
brick_f264a47aa32be5d595f83477572becf8/brick
total 180300
-rw-r-----. 2 1000070000 2001       56 Mar 24 12:11 auto.cnf
-rw-------. 2 1000070000 2001     1676 Mar 24 12:11 ca-key.pem
-rw-r--r--. 2 1000070000 2001     1075 Mar 24 12:11 ca.pem
-rw-r--r--. 2 1000070000 2001     1079 Mar 24 12:12 client-cert.
pem
-rw-------. 2 1000070000 2001     1680 Mar 24 12:12 client-key.p
em
-rw-r-----. 2 1000070000 2001      352 Mar 24 12:12 ib_buffer_po
ol
-rw-r-----. 2 1000070000 2001 12582912 Mar 24 12:20 ibdata1
-rw-r-----. 2 1000070000 2001 79691776 Mar 24 12:20 ib_logfile0
-rw-r-----. 2 1000070000 2001 79691776 Mar 24 12:11 ib_logfile1
-rw-r-----. 2 1000070000 2001 12582912 Mar 24 12:12 ibtmp1
drwxr-s---. 2 1000070000 2001     8192 Mar 24 12:12 mysql
-rw-r-----. 2 1000070000 2001        2 Mar 24 12:12 mysql-1-h4af
b.pid
drwxr-s---. 2 1000070000 2001     8192 Mar 24 12:12 performance_
schema
-rw-------. 2 1000070000 2001     1676 Mar 24 12:12 private_key.
pem
-rw-r--r--. 2 1000070000 2001      452 Mar 24 12:12 public_key.p
em
drwxr-s---. 2 1000070000 2001       62 Mar 24 12:20 sampledb
-rw-r--r--. 2 1000070000 2001     1079 Mar 24 12:11 server-cert.
pem
-rw-------. 2 1000070000 2001     1676 Mar 24 12:11 server-key.p
em
drwxr-s---. 2 1000070000 2001     8192 Mar 24 12:12 sys
```

你可以在这里看到MySQL数据库目录。 它使用GlusterFS作为后端存储，并作为绑定挂载给MySQL容器使用。 如果你检查OpenShift VM上的mount表，你将会看到GlusterFS的mount。

总结

在这里我们是在OpenShift之外创建了一个简单但功能强大的GlusterFS存储池。 该池可以独立于应用程序扩展和收缩。 该池的整个生命周期由一个简单的称为heketi的前端管理，你只需要在部署增长时进行手动干预。 对于日常配置操作，使用它的API与OpenShifts动态配置器交互，无需开发人员直接与基础架构团队进行交互。

o这就是我们如何将存储带入DevOps世界 - 无痛苦，并在OpenShift PaaS系统的开发人员工具中直接提供。

GlusterFS和OpenShift可跨越所有环境：裸机，虚拟机，私有和公共云（Azure，Google Cloud，AWS ...），确保应用程序可移植性，并避免云供应商锁定。

祝你愉快在容器中使用GlusterFS！

(c) 2017 Keith Tenzer

原文链接：https://keithtenzer.com/2017/03/24/storage-for-containers-using-gluster-part-ii/

Ceph的简要介绍

本文参考翻译自这篇文章的部分内容。

Ceph是一个开源的分布式对象，块和文件存储。该项目诞生于2003年，是塞奇·韦伊的博士论文的结果，然后在2006年在LGPL 2.1许可证发布。Ceph已经与Linux内核KVM集成，并且默认包含在许多GNU / Linux发行版中。

介绍

当前的工作负载和基础设施需要不同的数据访问方法（对象，块，文件），Ceph支持所有这些方法。它旨在具有可扩展性，并且没有单点故障。它是一款开源软件，可以在生产环境，通用硬件上运行。

RADOS （可靠的自动分布式对象存储）是Ceph的核心组件。RADOS对象和当今流行的对象之间存在着重要的区别，例如Amazon S3，OpenStack Swift或Ceph的RADOS对象网关提供的对象。从2005年到2010年，对象存储设备（OSD）成为一个流行的概念。这些OSD提供了强大的一致性，提供不同的接口，并且每个对象通常驻留在单个设备上。

在RADOS中有几种操作对象的方法：

- 在用C，C ++，Java，PHP和Python编写的应用程序中使用客户端库（librados）
- 使用命令行工具'rados'
- 使用与S3（Amazon）和Swift（OpenStack）兼容的现有API

RADOS是一个由Ceph节点组成的集群。有两种类型的节点：

- Ceph存储设备节点
- Ceph监控节点

每个Ceph存储设备节点运行一个或多个Ceph OSD守护进程，每个磁盘设备一个。OSD是一个Linux进程（守护进程），可处理与其分配的磁盘（HDD或SSD）相关的所有操作。所述OSD守护程序访问本地文件系统来存储数据和元数据，而不是直

接与磁盘通信。Ceph常用的文件系统是XFS，btrfs和ext4。每个OSD还需要一个日志，用于对RADOS对象进行原子更新。日志可能驻留在单独的磁盘上（通常是SSD以提高性能），但同一个磁盘可以被同一节点上的多个OSD使用。

该Ceph的监控节点上运行的单个Ceph的监控守护。Ceph Monitor守护程序维护集群映射的主副本。虽然Ceph集群可以与单个监控节点一起工作，但需要更多设备来确保高可用性。建议使用三个或更多Ceph Monitor节点，因为它们使用法定数量来维护集群映射。需要大多数Monitor来确认仲裁数，因此建议使用奇数个Monitor。例如，3个或4个Monitor都可以防止单个故障，而5个Monitor可以防止两个故障。

Ceph OSD守护进程和Ceph客户端可以感知群集，因此每个Ceph OSD守护进程都可以直接与其他Ceph OSD守护进程和Ceph监视器进行通信。此外，Ceph客户端可直接与Ceph OSD守护进程通信以读取和写入数据。

Ceph对象网关守护进程（radosgw） 提供了两个API：

- API与Amazon S3 RESTful AP的子集兼容
- API与OpenStack Swift API的子集兼容

如果RADOS和radosgw为客户提供对象存储服务，那么Ceph如何被用作块和文件存储？

Ceph中的分布式块存储（Ceph RDB）实现为对象存储顶部的薄层。Ceph RADOS块设备（RBD）存储分布在群集中多个Ceph OSD上的数据。RBD利用RADOS功能，如快照，复制和一致性。RBD使用Linux内核模块或librbd库与RADOS通信。此外，KVM管理程序可以利用librbd允许虚拟机访问Ceph卷。

Ceph文件系统（CephFS）是一个符合POSIX的文件系统，使用Ceph集群来存储其数据。所述Ceph的文件系统要求Ceph的集群中的至少一个Ceph的元数据服务器（MDS）。MDS处理所有文件操作，例如文件和目录列表，属性，所有权等。MDS利用RADOS对象来存储文件系统数据和属性。它可以水平扩展，因此您可以将更多的Ceph元数据服务器添加到您的群集中，以支持更多的文件系统操作客户端。

Kubernetes和Ceph

Kubernetes支持Ceph的块存储（Ceph RDB)和文件存储（CephFS）作为Kubernetes的持久存储后端。Kubernetes自带Ceph RDB的internal provisioner，可以配置动态提供，如果要使用CephFS作为动态存储提供，需要安装外置的

provisioner。

与Ceph相关的Kubernetes StorageClass的官方文档介绍

Volume Plugin	Internal Provisioner	Config Example
AWSElasticBlockStore	✓	AWS
AzureFile	✓	Azure File
AzureDisk	✓	Azure Disk
CephFS	-	-
Cinder	✓	OpenStack Cinder
FC	-	-
FlexVolume	-	-
Flocker	✓	-
GCEPersistentDisk	✓	GCE
Glusterfs	✓	Glusterfs
iSCSI	-	-
PhotonPersistentDisk	✓	-
Quobyte	✓	Quobyte
NFS	-	-
RBD	✓	Ceph RBD
VsphereVolume	✓	vSphere
PortworxVolume	✓	Portworx Volume
ScaleIO	✓	ScaleIO
StorageOS	✓	StorageOS
Local	-	Local

后续文档将介绍Kubernetes如何与Ceph RDB 和 CephFS集成。

用Helm托管安装Ceph集群并提供后端存储

本文翻译自Ceph官方文档，括号内的内容为注释。

安装

ceph-helm 项目可让你在Kubernetes 环境以托管方式部署Ceph . 本文档假定Kubernetes 环境已经可用。

当前的限制

- Public网络和Cluster网络必须是同一个网络
- 如果 storage class 用户标识不是admin, 则必须在Ceph集群中手动创建用户并在Kubernetes中创建其secret
- ceph-mgr只能运行1个replica

安装并使用Helm

可以按照此说明instructions安装Helm。

Helm通过从本地读取Kubernetes配置文件来查找Kubernetes集群; 确保文件已下载和且helm客户端可以访问。

Kubernetes群集必须配置并运行Tiller服务器，并且须将本地Helm客户端网络可达。查看init的Helm文档获取帮助。要在本地运行Tiller并将Helm连接到它，请运行如下命令（此命令会在Kubernetes集群部署一个tiller实例）：

```
$ helm init
```

ceph-helm项目默认使用本地的Helm repo来存储charts。要启动本地Helm repo服务器，请运行：

```
$ helm serve &
$ helm repo add local http://localhost:8879/charts
```

添加Ceph-Helm charts到本地repo

```
$ git clone https://github.com/ceph/ceph-helm
$ cd ceph-helm/ceph
$ make
```

配置Ceph集群

创建一个包含Ceph配置的ceph-overrides.yaml文件。这个文件可能存在于任何地方，本文档默认此文件在用户的home目录中。

```
$ cat ~/ceph-overrides.yaml
```

```yaml
network:
  public:   172.21.0.0/20
  cluster:   172.21.0.0/20

osd_devices:
  - name: dev-sdd
    device: /dev/sdd
    zap: "1"
  - name: dev-sde
    device: /dev/sde
    zap: "1"

storageclass:
  name: ceph-rbd
  pool: rbd
  user_id: k8s
```

注意 如果未设置日志（journal）设备，它将与device设备同位置。另ceph-helm/ceph/ceph/values.yaml文件包含所有可配置的选项。

创建Ceph 集群的namespace

默认情况下，ceph-helm组件在Kubernetes的ceph namespace中运行。如果要自定义，请自定义namespace的名称，默认namespace请运行：

```
$ kubectl create namespace ceph
```

配置RBAC权限

Kubernetes> = v1.6使RBAC成为默认的admission controller。ceph-helm要为每个组件提供RBAC角色和权限：

```
$ kubectl create -f ~/ceph-helm/ceph/rbac.yaml
```

rbac.yaml文件假定Ceph集群将部署在ceph命名空间中。

给Kubelet节点打标签

需要设置以下标签才能部署Ceph集群：

```
ceph-mon=enabled
ceph-mgr=enabled
ceph-osd=enabled
ceph-osd-device-<name>=enabled
```

ceph-osd-device-标签是基于我们的ceph-overrides.yaml中定义的osd_devices名称值创建的。从我们下面的例子中，我们将得到以下两个标签：ceph-osd-device-dev-sdb和ceph-osd-device-dev-sdc。

每个 Ceph Monitor节点:

```
$ kubectl label node <nodename> ceph-mon=enabled ceph-mgr=enable
d
```

每个 OSD node节点:

```
$ kubectl label node <nodename> ceph-osd=enabled ceph-osd-device
-dev-sdb=enabled ceph-osd-device-dev-sdc=enabled
```

Ceph 部署

运行helm install命令来部署Ceph：

```
$ helm install --name=ceph local/ceph --namespace=ceph -f ~/ceph
```

```
-overrides.yaml
NAME:   ceph
LAST DEPLOYED: Wed Oct 18 22:25:06 2017
NAMESPACE: ceph
STATUS: DEPLOYED

RESOURCES:
==> v1/Secret
NAME                    TYPE    DATA  AGE
ceph-keystone-user-rgw  Opaque  7     1s

==> v1/ConfigMap
NAME               DATA  AGE
ceph-bin-clients   2     1s
ceph-bin           24    1s
ceph-etc           1     1s
ceph-templates     5     1s

==> v1/Service
NAME      CLUSTER-IP      EXTERNAL-IP  PORT(S)   AGE
ceph-mon  None            <none>       6789/TCP  1s
ceph-rgw  10.101.219.239  <none>       8088/TCP  1s

==> v1beta1/DaemonSet
NAME               DESIRED  CURRENT  READY  UP-TO-DATE  AVAILABLE
  NODE-SELECTOR                                         AGE
ceph-mon           3        3        0      3           0
  ceph-mon=enabled                                      1s
ceph-osd-dev-sde   3        3        0      3           0
  ceph-osd-device-dev-sde=enabled,ceph-osd=enabled     1s
ceph-osd-dev-sdd   3        3        0      3           0
  ceph-osd-device-dev-sdd=enabled,ceph-osd=enabled     1s

==> v1beta1/Deployment
NAME                   DESIRED  CURRENT  UP-TO-DATE  AVAILABLE  A
GE
ceph-mds               1        1        1           0          1
s
ceph-mgr               1        1        1           0          1
s
ceph-mon-check         1        1        1           0          1
s
ceph-rbd-provisioner   2        2        2           0          1
s
ceph-rgw               1        1        1           0          1
s

==> v1/Job
NAME                                 DESIRED  SUCCESSFUL  AGE
ceph-mgr-keyring-generator           1        0           1s
ceph-mds-keyring-generator           1        0           1s
ceph-osd-keyring-generator           1        0           1s
ceph-rgw-keyring-generator           1        0           1s
```

```
ceph-mon-keyring-generator           1         0            1s
ceph-namespace-client-key-generator  1         0            1s
ceph-storage-keys-generator          1         0            1s

==> v1/StorageClass
NAME      TYPE
ceph-rbd   ceph.com/rbd
```

helm install的输出显示了将要部署的不同类型的资源。

将使用ceph-rbd-provisioner Pod创建ceph.com/rbd类型的名为ceph-rbd的StorageClass。这允许创建PVC时自动提供RBD。第一次挂载时，RBD设备将被格式化（format）。所有RBD设备都将使用ext4文件系统。ceph.com/rbd不支持fsType选项。默认情况下，RBD将使用镜像格式2和镜像分层特性。可以在values文件中覆盖以下storageclass的默认值：

```
storageclass:
  name: ceph-rbd
  pool: rbd
  user_id: k8s
  user_secret_name: pvc-ceph-client-key
  image_format: "2"
  image_features: layering
```

使用下面的命令检查所有Pod是否正常运行。这可能需要几分钟时间：

```
$ kubectl -n ceph get pods
NAME                                        READY     STATUS    REST
ARTS   AGE
ceph-mds-3804776627-976z9                   0/1       Pending   0
       1m
ceph-mgr-3367933990-b368c                   1/1       Running   0
       1m
ceph-mon-check-1818208419-0vkb7             1/1       Running   0
       1m
ceph-mon-cppdk                              3/3       Running   0
       1m
ceph-mon-t4stn                              3/3       Running   0
       1m
ceph-mon-vqzl0                              3/3       Running   0
       1m
ceph-osd-dev-sdd-6dphp                      1/1       Running   0
       1m
ceph-osd-dev-sdd-6w7ng                      1/1       Running   0
       1m
ceph-osd-dev-sdd-l80vv                      1/1       Running   0
       1m
ceph-osd-dev-sde-6dq6w                      1/1       Running   0
       1m
ceph-osd-dev-sde-kqt0r                      1/1       Running   0
       1m
ceph-osd-dev-sde-lp2pf                      1/1       Running   0
       1m
ceph-rbd-provisioner-2099367036-4prvt       1/1       Running   0
       1m
ceph-rbd-provisioner-2099367036-h9kw7       1/1       Running   0
       1m
ceph-rgw-3375847861-4wr74                   0/1       Pending   0
       1m
```

注意 因为我们没有用ceph-rgw = enabled或ceph-mds = enabled 给节点打标签（ceph对象存储特性需要ceph-rgw，cephfs特性需要ceph-mds），因此MDS和RGW Pod都处于pending状态，一旦其他Pod都在运行状态，请用如下命令从某个MON节点检查Ceph的集群状态：

```
$ kubectl -n ceph exec -ti ceph-mon-cppdk -c ceph-mon -- ceph -s
cluster:
  id:     e8f9da03-c2d2-4ad3-b807-2a13d0775504
  health: HEALTH_OK

services:
  mon: 3 daemons, quorum mira115,mira110,mira109
  mgr: mira109(active)
  osd: 6 osds: 6 up, 6 in

data:
  pools:   0 pools, 0 pgs
  objects: 0 objects, 0 bytes
  usage:   644 MB used, 5555 GB / 5556 GB avail
  pgs:
```

配置一个POD以便从Ceph申请使用一个持久卷

为~/ ceph-overwrite.yaml中定义的k8s用户创建一个密钥环，并将其转换为base64：

```
$ kubectl -n ceph exec -ti ceph-mon-cppdk -c ceph-mon -- bash
# ceph auth get-or-create-key client.k8s mon 'allow r' osd 'allow rwx pool=rbd'  | base64
QVFCLzdPaFoxeUxCRVJBQUVEVGdHcE9YU3BYMVBSdURHUEU0T0E9PQo=
# exit
```

编辑ceph namespace中存在的用户secret：

```
$ kubectl -n ceph edit secrets/pvc-ceph-client-key
```

将base64值复制到key位置的值并保存：:

```
apiVersion: v1
data:
  key: QVFCLzdPaFoxeUxCRVJBQUVEVGdHcE9YU3BYMVBSdURHUEU0T0E9PQo=
kind: Secret
metadata:
  creationTimestamp: 2017-10-19T17:34:04Z
  name: pvc-ceph-client-key
  namespace: ceph
  resourceVersion: "8665522"
  selfLink: /api/v1/namespaces/ceph/secrets/pvc-ceph-client-key
  uid: b4085944-b4f3-11e7-add7-002590347682
type: kubernetes.io/rbd
```

我们创建一个在default namespace中使用RBD的Pod。将用户secret从ceph namespace复制到default namespace：

```
$ kubectl -n ceph get secrets/pvc-ceph-client-key -o json | jq '
.metadata.namespace = "default"' | kubectl create -f -
secret "pvc-ceph-client-key" created
$ kubectl get secrets
NAME                  TYPE                                  DATA
      AGE
default-token-r43wl   kubernetes.io/service-account-token   3
      61d
pvc-ceph-client-key   kubernetes.io/rbd                     1
      20s
```

创建并初始化RBD池：

```
$ kubectl -n ceph exec -ti ceph-mon-cppdk -c ceph-mon -- ceph os
d pool create rbd 256
pool 'rbd' created
$ kubectl -n ceph exec -ti ceph-mon-cppdk -c ceph-mon -- rbd poo
l init rbd
```

重要 重要的 Kubernetes使用RBD内核模块将RBD映射到主机。Luminous需要CRUSH_TUNABLES 5（Jewel）。这些可调参数的最小内核版本是4.5。如果您的内核不支持这些可调参数，请运行ceph osd crush tunables hammer。

重要 由于RBD映射到主机系统上。主机需要能够解析由kube-dns服务管理的ceph-mon.ceph.svc.cluster.local名称。要获得kube-dns服务的IP地址，运行kubectl -n kube-system get svc/kube-dns。

创建一个PVC：

```
$ cat pvc-rbd.yaml
```

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: ceph-pvc
spec:
  accessModes:
   - ReadWriteOnce
  resources:
    requests:
       storage: 20Gi
  storageClassName: ceph-rbd
```

```
$ kubectl create -f pvc-rbd.yaml
persistentvolumeclaim "ceph-pvc" created
$ kubectl get pvc
NAME       STATUS    VOLUME
CAPACITY   ACCESSMODES   STORAGECLASS   AGE
ceph-pvc   Bound     pvc-1c2ada50-b456-11e7-add7-002590347682
20Gi       RWO           ceph-rbd        3s
```

检查集群上是否已创建RBD：

```
$ kubectl -n ceph exec -ti ceph-mon-cppdk -c ceph-mon -- rbd ls
kubernetes-dynamic-pvc-1c2e9442-b456-11e7-9bd2-2a4159ce3915
$ kubectl -n ceph exec -ti ceph-mon-cppdk -c ceph-mon -- rbd inf
o kubernetes-dynamic-pvc-1c2e9442-b456-11e7-9bd2-2a4159ce3915
rbd image 'kubernetes-dynamic-pvc-1c2e9442-b456-11e7-9bd2-2a4159
ce3915':
    size 20480 MB in 5120 objects
    order 22 (4096 kB objects)
    block_name_prefix: rbd_data.10762ae8944a
    format: 2
    features: layering
    flags:
    create_timestamp: Wed Oct 18 22:45:59 2017
```

创建一个使用此PVC的Pod：

```
$ cat pod-with-rbd.yaml
```

```
kind: Pod
apiVersion: v1
metadata:
  name: mypod
spec:
  containers:
    - name: busybox
      image: busybox
      command:
        - sleep
        - "3600"
      volumeMounts:
      - mountPath: "/mnt/rbd"
        name: vol1
  volumes:
    - name: vol1
      persistentVolumeClaim:
        claimName: ceph-pvc
```

```
$ kubectl create -f pod-with-rbd.yaml
pod "mypod" created
```

检查Pod：

```
$ kubectl get pods
NAME       READY     STATUS    RESTARTS   AGE
mypod      1/1       Running   0          17s
$ kubectl exec mypod -- mount | grep rbd
/dev/rbd0 on /mnt/rbd type ext4 (rw,relatime,stripe=1024,data=or
dered)
```

日志

可以通过kubectl logs [-f]命令访问OSD和Monitor日志。Monitors有多个日志记录流，每个流都可以从ceph-mon Pod中的容器访问。

在ceph-mon Pod中有3个容器运行：ceph-mon，相当于物理机上的ceph-mon.hostname.log，cluster-audit-log-tailer相当于物理机上的ceph.audit.log，cluster-log-tailer相当于物理机上的ceph.log或ceph -w。每个容器都可以通过--container或-c选项访问。例如，要访问cluster-tail-log，可以运行：

```
$ kubectl -n ceph logs ceph-mon-cppdk -c cluster-log-tailer
```

使用Ceph做持久化存储创建MySQL集群

本文中用到的 yaml 文件可以在 ../manifests/mariadb-cluster 目录下找到。

下面我们以部署一个高可用的 MySQL 集群为例，讲解如何使用 Ceph 做数据持久化，其中使用 StorageClass 动态创建 PV，Ceph 集群我们使用 kubernetes 集群外部的已有的集群，我们没有必要重新部署了。

在 1.4 以后，kubernetes 提供了一种更加方便的动态创建 PV 的方式；也就是说使用 StoragaClass 时无需预先创建固定大小的 PV，等待使用者创建 PVC 来使用；而是直接创建 PVC 即可分配使用。

使用 kubernetes 集群外部的 Ceph 存储

在部署 kubernetes 之前我们就已经有了 Ceph 集群，因此我们可以直接拿来用。但是 kubernetes 的所有节点（尤其是 master 节点）上依然需要安装 ceph 客户端。

```
yum install -y ceph-common
```

还需要将 ceph 的配置文件 `ceph.conf` 放在所有节点的 `/etc/ceph` 目录下。

Kubernetes 使用 ceph 存储需要用到如下配置：

- Monitors: Ceph montors 列表
- Path：作为挂载的根路径，默认是 /
- User：RADOS用户名，默认是 admin
- secretFile：keyring 文件路径，默认是 /etc/ceph/user.secret，我们 Ceph 集群提供的文件是 `ceph.client.admin.keyring` ，将在下面用到
- secretRef：Ceph 认证 secret 的引用，如果配置了将会覆盖 secretFile。
- readOnly：该文件系统是否只读。

Galera Cluster介绍

Galera是一个MySQL(也支持MariaDB，Percona)的同步多主集群软件。

从用户视角看，一组Galera集群可以看作一个具有多入口的MySQL库，用户可以同时从多个IP读写这个库。目前Galera已经得到广泛应用，例如Openstack中，在集群规模不大的情况下，稳定性已经得到了实践考验。真正的multi-master，即所有节点可以同时读写数据库。

详细步骤

以下步骤包括创建 Ceph 的配置 和 MySQL 的配置两部分。

配置 Ceph

关于 Ceph 的 yaml 文件可以在 ../manifest/mariadb-cluster 目录下找到。

1. 生成 Ceph secret

使用 Ceph 管理员提供给你的 `ceph.client.admin.keyring` 文件，我们将它放在了 `/etc/ceph` 目录下，用来生成 secret。

```
grep key /etc/ceph/ceph.client.admin.keyring |awk '{printf "%s", $NF}'|base64
```

将获得加密后的 key：`QVFDWDA2aFo5TG5TQnhBQVl1b0lUL2V3YlRSaEtwVEhPWkxvUlE9PQ==`，我们将在后面用到。

2. 创建租户namespace

创建 `galera-namespace.yaml` 文件内容为：

```yaml
apiVersion: v1
kind: Namespace
metadata:
  name: galera
```

3. 创建 Ceph secret

创建 `ceph-secret.yaml` 文件内容为：

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: ceph-secret
  namespace: galera
type: "kubernetes.io/rbd"
data:
  key: QVFDWDA2aFo5TG5TQnhBQVl1b0lUL2V3YlRSaEtwVEhPWkxvUlE9PQ==
```

4. 创建 StorageClass

创建 `ceph-class.yaml` 文件内容为：

```yaml
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
   name: ceph-web
provisioner: kubernetes.io/rbd
parameters:
  monitors: 172.28.7.98,172.28.7.99,172.28.7.100
  adminId: admin
  adminSecretName: ceph-secret
  adminSecretNamespace: galera
  pool: rbd #此处默认是rbd池，生产上建议自己创建存储池隔离
  userId: admin
  userSecretName: ceph-secret
```

此配置请参考 kubernetes 官方文档：https://kubernetes.io/docs/concepts/storage/persistent-volumes/#ceph-rbd

配置 MySQL

1. 创建 MySQL 配置文件

创建 `mysql-config.yaml` 文件内容为：

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: mysql-config-vol
  namespace: galera
  labels:
    app: mysql
data:
  mariadb.cnf: |
    [client]
    default-character-set = utf8
    [mysqld]
    character-set-server  = utf8
    collation-server      = utf8_general_ci
    # InnoDB optimizations
    innodb_log_file_size  = 64M
  galera.cnf: |
    [galera]
    user = mysql
    bind-address = 0.0.0.0
    # Optimizations
    innodb_flush_log_at_trx_commit = 0
    sync_binlog = 0
    expire_logs_days = 7
    # Required settings
    default_storage_engine = InnoDB
    binlog_format = ROW
    innodb_autoinc_lock_mode = 2
    query_cache_size = 0
    query_cache_type = 0
    # MariaDB Galera settings
    #wsrep_debug=ON
    wsrep_on=ON
    wsrep_provider=/usr/lib/galera/libgalera_smm.so
    wsrep_sst_method=rsync
    # Cluster settings (automatically updated)
    wsrep_cluster_address=gcomm://
    wsrep_cluster_name=galera
    wsrep_node_address=127.0.0.1
```

2. 创建 MySQL root 用户和密码

创建加密密码

```
$ echo -n jimmysong|base64
amltbXlzb25n
```

注意：一定要用-n 去掉换行符，不然会报错。

创建 **root** 用户

```
$ echo -n root |base64
cm9vdA==
```

创建 MySQL secret

创建 `mysql-secret.yaml` 文件内容为：

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: mysql-secrets
  namespace: galera
  labels:
    app: mysql
data:
  # Root password: changeit run  echo -n jimmysong|base64
  root-password: amltbXlzb25n
  # Root user: root
  root-user: cm9vdA==
```

3. 创建 yaml 配置文件

创建 MySQL 的 yaml 文件 `galera-mariadb.yaml` 内容为：

```yaml
apiVersion: v1
kind: Service
metadata:
  annotations:
    service.alpha.kubernetes.io/tolerate-unready-endpoints: "tru
e"
  name: mysql
  namespace: galera
  labels:
    app: mysql
    tier: data
spec:
  ports:
  - port: 3306
    name: mysql
  clusterIP: None
  selector:
    app: mysql
---
apiVersion: apps/v1beta1
kind: StatefulSet
```

```
metadata:
  name: mysql
  namespace: galera
spec:
  serviceName: "mysql"
  replicas: 3
  template:
    metadata:
      labels:
        app: mysql
        tier: data
      annotations:
        pod.beta.kubernetes.io/init-containers: '[
          {
            "name": "galera-init",
            "image": "harbor-001.jimmysong.io/library/k8s-galera
-init:latest",
            "args": ["-service=mysql"],
            "env": [
              {
                "name": "POD_NAMESPACE",
                "valueFrom": {
                  "fieldRef": { "apiVersion": "v1", "fieldPath":
"metadata.namespace" }
                }
              },
              {
                "name": "SAFE_TO_BOOTSTRAP",
                "value": "1"
              },
              {
                "name": "DEBUG",
                "value": "1"
              }
            ],
            "volumeMounts": [
              {
                "name": "config",
                "mountPath": "/etc/mysql/conf.d"
              },
              {
                "name": "data",
                "mountPath": "/var/lib/mysql"
              }
            ]
          }
        ]'
    spec:
      terminationGracePeriodSeconds: 10
      containers:
      - name: mysql
        image: harbor-001.jimmysong.io/library/mariadb:10.1
        imagePullPolicy: IfNotPresent
```

```
        ports:
        - containerPort: 3306
          name: mysql
        - containerPort: 4444
          name: sst
        - containerPort: 4567
          name: replication
        - containerPort: 4568
          name: ist
        env:
        - name: MYSQL_ROOT_PASSWORD
          valueFrom:
            secretKeyRef:
              name: mysql-secrets
              key: root-password
        - name: MYSQL_ROOT_USER
          valueFrom:
            secretKeyRef:
              name: mysql-secrets
              key: root-user
        - name: MYSQL_INITDB_SKIP_TZINFO
          value: "yes"
        livenessProbe:
          exec:
            command: ["sh", "-c", "mysql -u\"${MYSQL_ROOT_USER:-
root}\" -p\"${MYSQL_ROOT_PASSWORD}\" -e 'show databases;'"]
          initialDelaySeconds: 60
          timeoutSeconds: 5
        readinessProbe:
          exec:
            command: ["sh", "-c", "mysql -u\"${MYSQL_ROOT_USER:-
root}\" -p\"${MYSQL_ROOT_PASSWORD}\" -e 'show databases;'"]
          initialDelaySeconds: 20
          timeoutSeconds: 5
        volumeMounts:
        - name: config
          mountPath: /etc/mysql/conf.d
        - name: data
          mountPath: /var/lib/mysql
      volumes:
      - name: config
        configMap:
          name: mysql-config-vol
      imagePullSecrets:
        - name: "registrykey"
  volumeClaimTemplates:
  - metadata:
      name: data
      annotations:
        volume.beta.kubernetes.io/storage-class: "ceph-web" #引用
ceph  class 的类
    spec:
      accessModes: [ "ReadWriteOnce" ]
```

```
        resources:
          requests:
            storage: 3Gi
```

部署 MySQL 集群

在 `/etc/mariadb-cluster` 目录下执行：

```
kubectl create -f .
```

验证

存在 issue，参考 Error creating rbd image: executable file not found in $PATH#38923

问题记录

如果没有安装 `ceph-common` 的话，kubernetes 在创建 PVC 的时候会有如下报错信息：

```
Events:
  FirstSeen     LastSeen     Count     From                    SubObjec
tPath     Type          Reason            Message
  ---------     --------     -----     ----                    --------
-----     --------     ------            -------
  1h          12s         441     {persistentvolume-controller }
        Warning         ProvisioningFailed     Failed to provision
 volume with StorageClass "ceph-web": failed to create rbd image
: executable file not found in $PATH, command output:
```

检查 `kube-controller-manager` 的日志将看到如下错误信息：

```
journalctl -xe -u kube-controller-manager
... rbd_util.go:364] failed to create rbd image, output
... rbd.go:317] rbd: create volume failed, err: failed to create
 rbd image: executable file not found in $PATH, command output:
```

这是因为 `kube-controller-manager` 主机上没有安装 `ceph-common` 的缘故。

但是安装了 `ceph-common` 之后依然有问题：

```
Sep  4 15:25:36 bj-xg-oam-kubernetes-001 kube-controller-manager
: W0904 15:25:36.032128   13211 rbd_util.go:364] failed to creat
e rbd image, output
Sep  4 15:25:36 bj-xg-oam-kubernetes-001 kube-controller-manager
: W0904 15:25:36.032201   13211 rbd_util.go:364] failed to creat
e rbd image, output
Sep  4 15:25:36 bj-xg-oam-kubernetes-001 kube-controller-manager
: W0904 15:25:36.032252   13211 rbd_util.go:364] failed to creat
e rbd image, output
Sep  4 15:25:36 bj-xg-oam-kubernetes-001 kube-controller-manager
: E0904 15:25:36.032276   13211 rbd.go:317] rbd: create volume f
ailed, err: failed to create rbd image: fork/exec /usr/bin/rbd:
invalid argument, command output:
```

该问题尚未解决，参考 Error creating rbd image: executable file not found in $PATH#38923

从日志记录来看追查到 `pkg/volume/rbd/rbd.go` 的 `func (r *rbdVolumeProvisioner) Provision() (*v1.PersistentVolume, error) {` 方法对 `ceph-class.yaml` 中的参数进行了验证和处理后调用了 `pkg/volume/rbd/rdb_utils.go` 文件第 344 行 `CreateImage` 方法（kubernetes v1.6.1版本）：

```
func (util *RBDUtil) CreateImage(p *rbdVolumeProvisioner) (r *v1.
RBDVolumeSource, size int, err error) {
    var output []byte
    capacity := p.options.PVC.Spec.Resources.Requests[v1.Resourc
eName(v1.ResourceStorage)]
    volSizeBytes := capacity.Value()
    // convert to MB that rbd defaults on
    sz := int(volume.RoundUpSize(volSizeBytes, 1024*1024))
    volSz := fmt.Sprintf("%d", sz)
    // rbd create
    l := len(p.rbdMounter.Mon)
    // pick a mon randomly
    start := rand.Int() % l
    // iterate all monitors until create succeeds.
    for i := start; i < start+l; i++ {
        mon := p.Mon[i%l]
        glog.V(4).Infof("rbd: create %s size %s using mon %s, po
ol %s id %s key %s", p.rbdMounter.Image, volSz, mon, p.rbdMounter
.Pool, p.rbdMounter.adminId, p.rbdMounter.adminSecret)
        output, err = p.rbdMounter.plugin.execCommand("rbd",
            []string{"create", p.rbdMounter.Image, "--size", vol
Sz, "--pool", p.rbdMounter.Pool, "--id", p.rbdMounter.adminId, "
-m", mon, "--key=" + p.rbdMounter.adminSecret, "--image-format",
"1"})
        if err == nil {
            break
        } else {
            glog.Warningf("failed to create rbd image, output %v"
, string(output))
        }
    }

    if err != nil {
        return nil, 0, fmt.Errorf("failed to create rbd image: %
v, command output: %s", err, string(output))
    }

    return &v1.RBDVolumeSource{
        CephMonitors: p.rbdMounter.Mon,
        RBDImage:     p.rbdMounter.Image,
        RBDPool:      p.rbdMounter.Pool,
    }, sz, nil
}
```

该方法调用失败。

参考

https://github.com/kubernetes/examples/blob/master/staging/volumes/cephfs/README.md

k8s-ceph-statefulsets-storageclass-nfs 动态卷有状态应用实践

Kubernetes persistent storage with Ceph

https://kubernetes.io/docs/concepts/storage/persistent-volumes/#ceph-rbd

Error creating rbd image: executable file not found in $PATH#38923

OpenEBS

OpenEBS是一款使用Go语言编写的基于容器的块存储开源软件。OpenEBS使得在容器中运行关键性任务和需要数据持久化的负载变得更可靠。

OpenEBS由CloudByte研发，这是一家专业做容器化存储的公司，OpenEBS是其一款开源产品，CloudByte将其在企业级容器存储的经验付诸到该项目中。这个项目的愿景也很简单，就是让需要持久化存储的工作负载中的存储服务能够直接集成在环境中，存储服务可以自动管理，将存储的细节隐藏起来，就像存储系统是另一套基础架构一样。

我们知道AWS中提供了EBS（Elastic Block Storage），适用于 Amazon EC2 的持久性块存储，可以满足要求最苛刻的应用程序在功能和性能方面的要求，OpenEBS即其开源实现。

简介

使用OpenEBS，你可以将有持久化数据的容器，像对待其他普通容器一样来对待。OpenEBS本身也是通过容器来部署的，支持Kubernetes、Swarm、Mesos、Rancher编排调度，存储服务可以分派给每个pod、应用程序、集群或者容器级别，包括：

- 跨节点的数据持久化
- 跨可用区和云厂商的数据同步
- 使用商业硬件和容器引擎来提供高可扩展的块存储
- 与容器编排引擎集成，开发者的应用程序可以自动的配置OpenEBS
- 基于CloudByte在BSD的容器化经验，为用户提供OpenEBS的QoS保证

架构

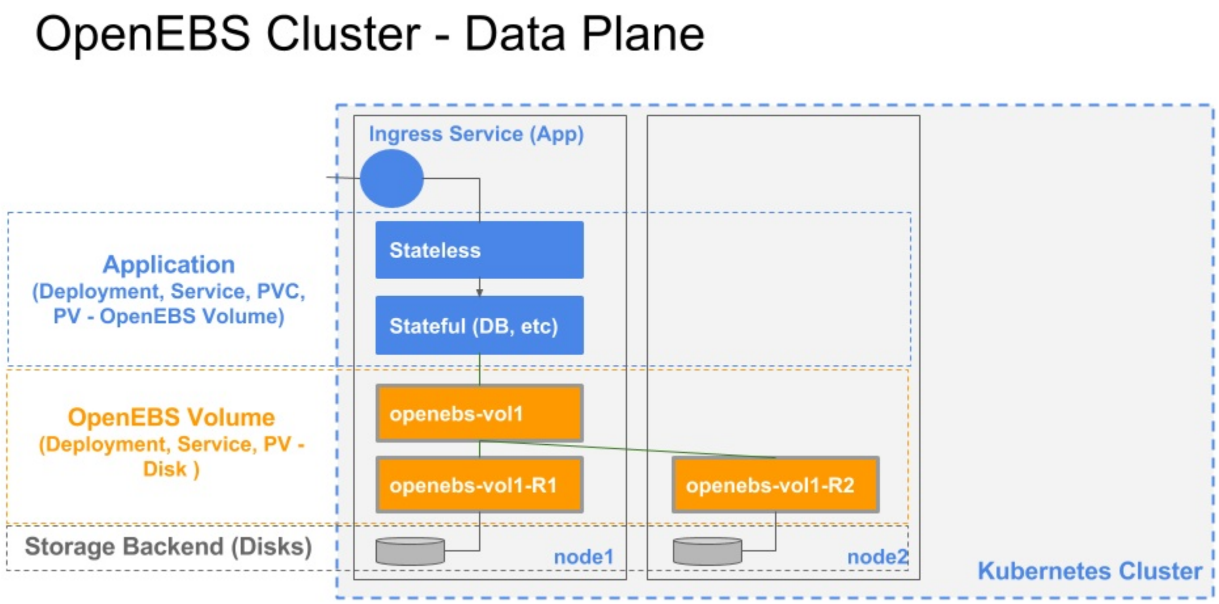
OpenEBS存储控制器本身就运行在容器中。OpenEBS Volume由一个或多个以微服务方式运行的容器组成。这种存储控制器功能基于微服务架构——每个卷的数据由其自己的一组容器来提供，而不是由一个统一的同时为多个卷提供控制的，单体（monolithic）存储控制器来提供。这就是OpenEBS与传统存储设备的本质区别。

OpenEBS的架构可以分为数据平面（Data Plane）和控制平面（Control Plane）两部分：

- 数据平面：为应用程序提供数据存储
- 控制平面：管理OpenEBS卷容器，这通常会用到容器编排软件的功能

数据平面

下图是OpenEBS对应在Kubernetes集群上部署的架构图。其中，黄色部分是OpenEBS持久化存储卷，通过Kubernetes的PV来创建，使用iSCSI来实现，数据保存在node节点上或者云中的卷（如EBS、GPD等），这取决于您的集群部署在哪里。OpenEBS的卷完全独立于用户的应用的生命周期来管理，这也是Kuberentes中的PV的基本思路。



图片 - *OpenEBS*集群数据平面（图片来自*https://github.com/openebs/openebs/blob/master/contribute/design/README.md*）

OpenEBS卷为容器提供持久化存储，具有针对系统故障的弹性，更快地访问存储，快照和备份功能。此外，它还提供了监控使用情况和执行QoS策略的机制。

存储数据的磁盘称为存储后端，可以是主机目录，附加块设备或远程磁盘。每个OpenEBS卷包含一个iSCSI目标容器（在上图中表示为openebs-vol1）和一个或多个副本容器（openebs-vol1-R1和openebs-vol1-R2）。

应用程序pod通过iSCSI目标容器访问存储，iSCSI目标容器将数据复制到其所有副本。在发生节点故障时，iSCSI目标容器将从剩余的其中一个在线节点上启动，并通过连接到可用副本容器来提供数据。

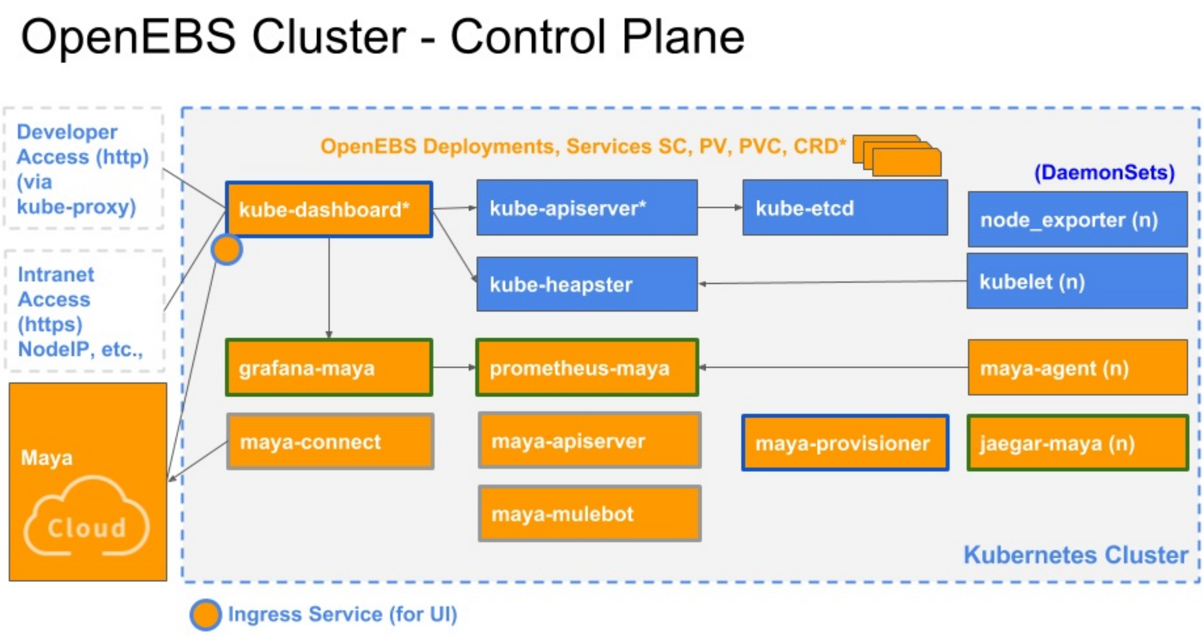
源码

该部分的实现包括两个容器：

- openebs/jiva：存储控制功能，包括复制逻辑
- openebs/gotgt：由openebs/jiva使用的iSCSI目标功能

控制平面

OpenEBS控制平面又叫做存储编排或maya。目的是为了创建超融合的OpenEBS，将其挂载到如Kubernetes、Swarm、Nomad等容器编排调度引擎上，用来扩展特定的容器编排系统提供的存储功能。



图片 - *OpenEBS*集群的控制平面*(*图片来自
https://github.com/openebs/openebs/blob/master/contribute/design/README.md
)

OpenEBS的控制平面也是基于微服务的，它的服务可以分成以下几个部分：

- 容器编排插件，用于增加强容器编排框架的功能：
  - **Kubernetes**动态配置：openebs-provisioner
  - **Kubernetes-dashboard**：openebs-dashboard
  - 扩展的**schema**：基于Kubernetes的CRD（自定义资源类型），存储OpenEBS相关的配置数据
- 集群服务，提供OpenEBS特定的存储智能，如：
  - **maya-apiserver**：包含执行卷操作的API，可将请求转换为容器编排系统特定的操作
  - **maya-mulebot**：使用收集的信息来建议优化的布局和事件处理提示
  - **maya-connect**：允许将监控数据上传到 `maya-cloud` ，以便进一步进行存储访问模式分析
- 节点服务，提供OpenEBS特定的随kubelet一起运行的存储智能，如：
  - **maya-agent**：包括存储管理功能

通过使用prometheus、heapster、grafana和jaegar进行上述服务，可以添加监控和跟踪功能。

源码

- openebs/maya：所有特定的二进制代码（非插件）都存储在这个仓库中，比如 `maya-apiserver` 、 `maya-agent` 、 `maya-mulebot` 、 `maya-connect` 、 `mayactl` 等等。
- openebs-dashboard：kubernetes-dashboard项目的分支，扩展了存储功能。
- openebs-provisioner：来自Kubernetes孵化器项目的OpenEBS K8s Provisioner。

参考

- https://www.openebs.io/
- https://github.com/openebs/openebs
- Data Scientists adopting tools and solutions that allow them to focus more on Data Science and less on the infrastructure around them

使用OpenEBS做持久化存储

本文将指导您如何在Kubernetes集群上安装OpenEBS作为持久化存储。

我们将使用Operator的方式来安装OpenEBS，安装之前需要先确认您的节点上已经安装了iSCSI。

先决条件

OpenEBS依赖与iSCSI做存储管理，因此需要先确保您的集群上已有安装OpenEBS。

注意：如果您使用kubeadm，容器方式安装的kublet，那么其中会自带iSCSI，不需要再手动安装，如果是直接使用二进制形式在裸机上安装的kubelet，则需要自己安装iSCSI。

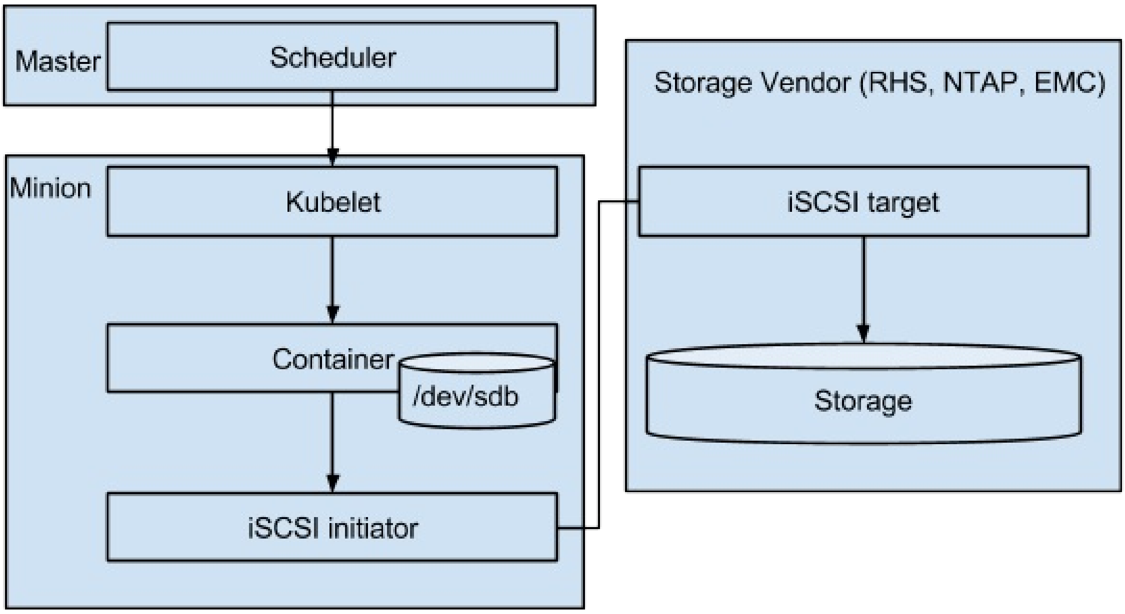
iSCSI(Internet Small Computer System Interface 互联网小型计算机系统接口)是一种基于TCP/IP 的协议，用来建立和管理IP存储设备、主机和客户机等之间的相互连接，并创建存储区域网络（SAN）。SAN 使得SCSI 协议应用于高速数据传输网络成为可能，这种传输以数据块级别（block-level）在多个数据存储网络间进行。SCSI 结构基于C/S模式，其通常应用环境是：设备互相靠近，并且这些设备由SCSI 总线连接。

OpenEBS需要使用iSCSI作为存储协议，而CentOS上默认是没有安装该软件的，因此我们需要手动安装。

iSCSI中包括两种类型的角色：

- **target**：用来提供存储（server）
- **initiator**：使用存储的客户端（client）

下图在Kubernetes中使用iSCSI的架构图（图片来源：http://rootfs.github.io/iSCSI-Kubernetes/）。



图片 - *Kubernetes iSCSI*架构

安装iSCSI服务十分简单，不需要额外的配置，只要安装后启动服务即可。

在每个node节点上执行下面的命令：

```
yum -y install iscsi-initiator-utils
systemctl enable iscsid
systemctl start iscsid
```

快速开始

使用Operator运行OpenEBS服务：

```
wget https://raw.githubusercontent.com/openebs/openebs/master/k8
s/openebs-operator.yaml
kubectl apply -f openebs-operator.yaml
```

使用默认或自定义的storageclass：

```
wget https://raw.githubusercontent.com/openebs/openebs/master/k8
s/openebs-storageclasses.yaml
kubectl apply -f openebs-storageclasses.yaml
```

用到的镜像有：

- openebs/m-apiserver:0.5.1-RC1
- openebs/openebs-k8s-provisioner:0.5.1-RC2
- openebs/jiva:0.5.1-RC1
- openebs/m-exporter:0.5.0

测试

下面使用OpenEBS官方文档中的示例，安装Jenkins测试

```
wget https://raw.githubusercontent.com/openebs/openebs/master/k8
s/demo/jenkins/jenkins.yml
kubectl apply -f jenkins.yml
```

查看PV和PVC

```
$ kubectl get pv
NAME                                       CAPACITY    ACCESS MOD
ES    RECLAIM POLICY    STATUS     CLAIM                    STORAG
ECLASS        REASON     AGE
pvc-8e203e86-f1e5-11e7-aa47-f4e9d49f8ed0   5G          RWO
     Delete            Bound      default/jenkins-claim     openeb
s-standard               1h

$ kubectl get pvc
kubectl get pvc
NAME                 STATUS     VOLUME
       CAPACITY    ACCESS MODES    STORAGECLASS        AGE
jenkins-claim       Bound      pvc-8e203e86-f1e5-11e7-aa47-f4e9d49f
8ed0    5G          RWO            openebs-standard    1h
```

查看Jenkins pod

```
Events:
  Type     Reason                 Age                From
          Message
  ----     ------                 ----               ----
          -------
  Warning  FailedScheduling       29m (x2 over 29m)  default-sch
eduler     PersistentVolumeClaim is not bound: "jenkins-claim" (
repeated 3 times)
  Normal   Scheduled              29m                default-sch
eduler     Successfully assigned jenkins-668dfbd847-vhg4c to 17
2.20.0.115
  Normal   SuccessfulMountVolume  29m                kubelet, 17
2.20.0.115  MountVolume.SetUp succeeded for volume "default-toke
n-3l9f0"
  Warning  FailedMount            27m                kubelet, 17
2.20.0.115  Unable to mount volumes for pod "jenkins-668dfbd847-
vhg4c_default(8e2ad467-f1e5-11e7-aa47-f4e9d49f8ed0)": timeout ex
pired waiting for volumes to attach/mount for pod "default"/"jen
kins-668dfbd847-vhg4c". list of unattached/unmounted volumes=[je
nkins-home]
  Warning  FailedSync             27m                kubelet, 17
2.20.0.115  Error syncing pod
  Normal   SuccessfulMountVolume  26m                kubelet, 17
2.20.0.115  MountVolume.SetUp succeeded for volume "pvc-8e203e86
-f1e5-11e7-aa47-f4e9d49f8ed0"
  Normal   Pulling                26m                kubelet, 17
2.20.0.115  pulling image "harbor-001.jimmysong.io/library/jenki
ns:lts"
  Normal   Pulled                 26m                kubelet, 17
2.20.0.115  Successfully pulled image "harbor-001.jimmysong.io/l
ibrary/jenkins:lts"
  Normal   Created                26m                kubelet, 17
2.20.0.115  Created container
  Normal   Started                26m                kubelet, 17
2.20.0.115  Started container
```

启动成功。Jenkins配置使用的是**NodePort**方式访问，现在访问集群中任何一个节点的Jenkins service的NodePort即可。

存储策略

OpenEBS的存储策略使用StorageClaass实现，包括如下的StorageClass：

- openebs-cassandra
- openebs-es-data-sc
- openebs-jupyter

- openebs-kafka
- openebs-mongodb
- openebs-percona
- openebs-redis
- openebs-standalone
- openebs-standard
- openebs-zk

每个Storage Class都对应与某种应用的存储模式，使用方式请参考OpenEBS Storage Policies。

参考

- OpenEBS Documentation
- CentOS 7.x 下配置iSCSI网络存储
- Configure iSCSI Initiator
- RHEL7: Configure a system as either an iSCSI target or initiator that persistently mounts an iSCSI target.

Rook

Rook是一款云原生环境下的开源分布式存储编排系统，目前已进入CNCF孵化。Rook的官方网站是https://rook.io。

Rook是什么？

Rook将分布式存储软件转变为自我管理，自我缩放和自我修复的存储服务。它通过自动化部署，引导、配置、供应、扩展、升级、迁移、灾难恢复、监控和资源管理来实现。 Rook使用基础的云原生容器管理、调度和编排平台提供的功能来履行其职责。

Rook利用扩展点深入融入云原生环境，为调度、生命周期管理、资源管理、安全性、监控和用户体验提供无缝体验。

Rook现在处于alpha状态，并且最初专注于在Kubernetes之上运行Ceph。Ceph是一个分布式存储系统，提供文件、数据块和对象存储，可以部署在大型生产集群中。Rook计划在未来的版本中增加对除Ceph之外的其他存储系统以及Kubernetes之外的其他云原生环境的支持。

部署

可以使用helm或直接用yaml文件两种方式来部署rook operator。

使用**helm**部署

```
helm init -i jimmysong/kubernetes-helm-tiller:v2.8.1
helm repo add rook-alpha https://charts.rook.io/alpha
helm install rook-alpha/rook --name rook --namespace rook-system
```

直接使用**yaml**文件部署

```
kubectl apply -f rook-operator.yaml
```

不论使用那种方式部署的rook operator，都会在rook-agent中看到rook-agent用户无法列出集群中某些资源的错误，可以通过为rook-agent的分配 `cluster-admin` 权限临时解决，详见Issue 1472。

使用如下yanl文件创建一个 `ClusterRoleBinding` 并应用到集群中。

```yaml
kind: ClusterRoleBinding
apiVersion: rbac.authorization.k8s.io/v1beta1
metadata:
    name: rookagent-clusterrolebinding
subjects:
  - kind: ServiceAccount
    name: rook-agent
    namespace: rook-system
roleRef:
    kind: ClusterRole
    name: cluster-admin
    apiGroup: ""
```

部署rook cluster

创建完rook operator后，我们再部署rook cluster。

`rook-cluster.yaml` 配置如下：

```yaml
apiVersion: v1
kind: Namespace
metadata:
  name: rook
---
apiVersion: rook.io/v1alpha1
kind: Cluster
metadata:
  name: rook
  namespace: rook
spec:
  versionTag: v0.6.2
  dataDirHostPath: /var/lib/rook
  storage:
    useAllNodes: true
    useAllDevices: false
    storeConfig:
      storeType: bluestore
      databaseSizeMB: 1024
      journalSizeMB: 1024
```

注意：需要手动指定 `versionTag` ，因为该镜像repo中没有 `latest` 标签，如不指定的话Pod将出现镜像拉取错误。

执行下面的命令部署rook集群。

```
kubectl apply -f rook-cluster.yaml
```

rook集群运行在 `rook` namespace下，查看rook集群中的pod：

```
$ kubectl -n rook get pod
NAME                             READY     STATUS    RESTARTS
AGE
rook-api-848df956bf-q6zf2        1/1       Running   0
4m
rook-ceph-mgr0-cfccfd6b8-cpk5p   1/1       Running   0
4m
rook-ceph-mon0-t496l             1/1       Running   0
6m
rook-ceph-mon1-zcn7v             1/1       Running   0
5m
rook-ceph-mon3-h97qx             1/1       Running   0
3m
rook-ceph-osd-557tn              1/1       Running   0
4m
rook-ceph-osd-74frb              1/1       Running   0
4m
rook-ceph-osd-zf7rg              1/1       Running   1
4m
rook-tools                       1/1       Running   0
2m
```

部署StorageClass

StorageClass rook-block的yaml文件（rook-storage.yaml）如下：

```yaml
apiVersion: rook.io/v1alpha1
kind: Pool
metadata:
  name: replicapool
  namespace: rook
spec:
  replicated:
    size: 1
  # For an erasure-coded pool, comment out the replication size above and uncomment the following settings.
  # Make sure you have enough OSDs to support the replica size or erasure code chunks.
  #erasureCoded:
  #  dataChunks: 2
  #  codingChunks: 1
---
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
   name: rook-block
provisioner: rook.io/block
parameters:
  pool: replicapool
  # Specify the Rook cluster from which to create volumes.
  # If not specified, it will use `rook` as the name of the cluster.
  # This is also the namespace where the cluster will be
  clusterName: rook
  # Specify the filesystem type of the volume. If not specified, it will use `ext4`.
  # fstype: ext4
```

我们在下面的示例中将使用rook-block这个StorageClass来创建PV。

工具

部署rook操作工具pod，该工具pod的yaml文件（rook-tools.yaml）如下：

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: rook-tools
  namespace: rook-system
spec:
  dnsPolicy: ClusterFirstWithHostNet
  serviceAccountName: rook-operator
  containers:
  - name: rook-tools
    image: rook/toolbox:master
    imagePullPolicy: IfNotPresent
    env:
      - name: ROOK_ADMIN_SECRET
        valueFrom:
          secretKeyRef:
            name: rook-ceph-mon
            key: admin-secret
    securityContext:
      privileged: true
    volumeMounts:
      - mountPath: /dev
        name: dev
      - mountPath: /sys/bus
        name: sysbus
      - mountPath: /lib/modules
        name: libmodules
      - name: mon-endpoint-volume
        mountPath: /etc/rook
  hostNetwork: false
  volumes:
    - name: dev
      hostPath:
        path: /dev
    - name: sysbus
      hostPath:
        path: /sys/bus
    - name: libmodules
      hostPath:
        path: /lib/modules
    - name: mon-endpoint-volume
      configMap:
        name: rook-ceph-mon-endpoints
        items:
        - key: endpoint
          path: mon-endpoints
```

`ConfigMap` 和 `Secret` 中的配置项内容是自定义的。

使用下面的命令部署工具pod：

```
kubectl apply -f rook-tools.yaml
```

这是一个独立的pod，没有使用其他高级的controller来管理，我们将它部署在 `rook-system` 的namespace下。

```
kubectl -n rook exec -it rook-tools bash
```

使用下面的命令查看rook集群状态。

```
$ rookctl status
OVERALL STATUS: OK

USAGE:
TOTAL       USED       DATA      AVAILABLE
37.95 GiB   1.50 GiB   0 B       36.45 GiB

MONITORS:
NAME             ADDRESS                 IN QUORUM    STATUS
rook-ceph-mon0   10.254.162.99:6790/0    true         UNKNOWN

MGRs:
NAME             STATUS
rook-ceph-mgr0   Active

OSDs:
TOTAL     UP        IN        FULL      NEAR FULL
1         1         1         false     false

PLACEMENT GROUPS (0 total):
STATE     COUNT
none

$ ceph df
GLOBAL:
    SIZE       AVAIL      RAW USED     %RAW USED
    38861M     37323M        1537M          3.96
POOLS:
    NAME     ID     USED     %USED     MAX AVAIL     OBJECTS
```

示例

官方提供了使用rook作为典型的LAMP（Linux + Apache + MySQL + PHP）应用Wordpress的存储后端的示例的yaml文件 `mysql.yaml` 和 `wordpress.yaml` ，使用下面的命令创建。

```
kubectl apply -f mysql.yaml
kubectl apply -f wordpress.yaml
```

Wordpress要依赖于MySQL，所以要先创建MySQL。

在创建wordpress的时候可能遇到该错误rook flexvolume failing to attach volumes #1147，该问题尚未解决。

清理

如果使用helm部署，则执行下面的命令：

```
helm delete --purge rook
helm delete daemonset rook-agent
```

如果使用yaml文件直接部署，则使用 `kubectl delete -f` 加当初使用的yaml文件即可删除集群。

参考

- Operator Helm Chart
- Creating Rook Clusters

NFS（Network File System）网络文件系统

NFS（Network File System）即网络文件系统，是FreeBSD支持的文件系统中的一种，它允许网络中的计算机之间通过TCP/IP网络共享资源。在NFS的应用中，本地NFS的客户端应用可以透明地读写位于远端NFS服务器上的文件，就像访问本地文件一样。在Linux系统中，NFS也作为一种简单的网络共享文件系统而存在。

利用NFS动态提供Kubernetes后端存储卷

本文翻译自nfs-client-provisioner的说明文档，本文将介绍使用nfs-client-provisioner这个应用，利用NFS Server给Kubernetes作为持久存储的后端，并且动态提供PV。前提条件是有已经安装好的NFS服务器，并且NFS服务器与Kubernetes的Slave节点都能网络连通。 所有下文用到的文件来自于 `git clone https://github.com/kubernetes-incubator/external-storage.git` 的nfs-client目录。

nfs-client-provisioner

nfs-client-provisioner 是一个Kubernetes的简易NFS的外部provisioner，本身不提供NFS，需要现有的NFS服务器提供存储

- PV以 `${namespace}-${pvcName}-${pvName}` 的命名格式提供（在NFS服务器上）
- PV回收的时候以 `archieved-${namespace}-${pvcName}-${pvName}` 的命名格式（在NFS服务器上）

安装部署

- 修改deployment文件并部署 `deploy/deployment.yaml`

需要修改的地方只有NFS服务器所在的IP地址（10.10.10.60），以及NFS服务器共享的路径（ `/ifs/kubernetes` ），两处都需要修改为你实际的NFS服务器和共享目录

```yaml
kind: Deployment
apiVersion: extensions/v1beta1
metadata:
  name: nfs-client-provisioner
spec:
  replicas: 1
  strategy:
    type: Recreate
  template:
    metadata:
      labels:
        app: nfs-client-provisioner
    spec:
      serviceAccountName: nfs-client-provisioner
      containers:
        - name: nfs-client-provisioner
          image: quay.io/external_storage/nfs-client-provisioner:
latest
          volumeMounts:
            - name: nfs-client-root
              mountPath: /persistentvolumes
          env:
            - name: PROVISIONER_NAME
              value: fuseim.pri/ifs
            - name: NFS_SERVER
              value: 10.10.10.60
            - name: NFS_PATH
              value: /ifs/kubernetes
      volumes:
        - name: nfs-client-root
          nfs:
            server: 10.10.10.60
            path: /ifs/kubernetes
```

- 修改StorageClass文件并部署 `deploy/class.yaml`

此处可以不修改，或者修改provisioner的名字，需要与上面的deployment的PROVISIONER_NAME名字一致。

```yaml
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: managed-nfs-storage
provisioner: fuseim.pri/ifs
```

授权

如果您的集群启用了RBAC，或者您正在运行OpenShift，则必须授权provisioner。如果你在非默认的“default”名称空间/项目之外部署，可以编辑 `deploy/auth/clusterrolebinding.yaml` 或编辑`oadm policy“指令。

如果启用了RBAC

需要执行如下的命令来授权。

```
$ kubectl create -f deploy/auth/serviceaccount.yaml
serviceaccount "nfs-client-provisioner" created
$ kubectl create -f deploy/auth/clusterrole.yaml
clusterrole "nfs-client-provisioner-runner" created
$ kubectl create -f deploy/auth/clusterrolebinding.yaml
clusterrolebinding "run-nfs-client-provisioner" created
$ kubectl patch deployment nfs-client-provisioner -p '{"spec":{"
template":{"spec":{"serviceAccount":"nfs-client-provisioner"}}}}'
```

测试

测试创建PVC

- `kubectl create -f deploy/test-claim.yaml`

测试创建POD

- `kubectl create -f deploy/test-pod.yaml`

在NFS服务器上的共享目录下的卷子目录中检查创建的NFS PV卷下是否有"SUCCESS" 文件。

删除测试POD

- `kubectl delete -f deploy/test-pod.yaml`

删除测试PVC

- `kubectl delete -f deploy/test-claim.yaml`

在NFS服务器上的共享目录下查看NFS的PV卷回收以后是否名字以archived开头。

我的示例

- NFS服务器配置

```
# cat /etc/exports


/media/docker        *(no_root_squash,rw,sync,no_subtree_check)
```

- nfs-deployment.yaml示例

NFS服务器的地址是ubuntu-master,共享出来的路径是/media/docker，其他不需要修改。

```
# cat nfs-deployment.yaml
```

```
kind: Deployment
apiVersion: extensions/v1beta1
metadata:
  name: nfs-client-provisioner
spec:
  replicas: 1
  strategy:
    type: Recreate
  template:
    metadata:
      labels:
        app: nfs-client-provisioner
    spec:
      serviceAccountName: nfs-client-provisioner
      containers:
        - name: nfs-client-provisioner
          image: quay.io/external_storage/nfs-client-provisioner:
latest
          volumeMounts:
            - name: nfs-client-root
              mountPath: /persistentvolumes
          env:
            - name: PROVISIONER_NAME
              value: fuseim.pri/ifs
            - name: NFS_SERVER
              value: ubuntu-master
            - name: NFS_PATH
              value: /media/docker
      volumes:
        - name: nfs-client-root
          nfs:
            server: ubuntu-master
            path: /media/docker
```

- StorageClass示例

可以修改Class的名字，我的改成了default。

```
# cat class.yaml


apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: default
provisioner: fuseim.pri/ifs
```

- 查看StorageClass

```
# kubectl get sc
NAME              PROVISIONER                AGE
default           fuseim.pri/ifs             2d
```

- 设置这个default名字的SC为Kubernetes的默认存储后端

```
# kubectl patch storageclass default -p '{"metadata": {"annotati
ons":{"storageclass.kubernetes.io/is-default-class":"true"}}}'
storageclass.storage.k8s.io "default" patched
# kubectl get sc
NAME                PROVISIONER                AGE
default (default)   fuseim.pri/ifs             2d
```

- 测试创建PVC

查看pvc文件

```
# cat test-claim.yaml


kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: test-claim
spec:
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 1Mi
```

创建PVC

```
# kubectl apply -f test-claim.yaml
persistentvolumeclaim "test-claim" created
root@Ubuntu-master:~/kubernetes/nfs# kubectl get pvc|grep test
test-claim                    Bound     pvc-fe3cb938-3f15-11e8-b61
d-08002795cb26    1Mi         RWX           default        10s
# kubectl get pv|grep test
pvc-fe3cb938-3f15-11e8-b61d-08002795cb26    1Mi        RWX
     Delete            Bound     default/test-claim
    default                    58s
```

- 启动测试POD

POD文件如下，作用就是在test-claim的PV里touch一个SUCCESS文件。

```
# cat test-pod.yaml


kind: Pod
apiVersion: v1
metadata:
  name: test-pod
spec:
  containers:
  - name: test-pod
    image: gcr.io/google_containers/busybox:1.24
    command:
      - "/bin/sh"
    args:
      - "-c"
      - "touch /mnt/SUCCESS && exit 0 || exit 1"
    volumeMounts:
      - name: nfs-pvc
        mountPath: "/mnt"
  restartPolicy: "Never"
  volumes:
    - name: nfs-pvc
      persistentVolumeClaim:
        claimName: test-claim
```

启动POD，一会儿POD就是completed状态，说明执行完毕。

```
# kubectl apply -f test-pod.yaml
pod "test-pod" created
kubectl get pod|grep test
test-pod                                                    0/1
    Completed   0          40s
```

我们去NFS共享目录查看有没有SUCCESS文件。

```
# cd default-test-claim-pvc-fe3cb938-3f15-11e8-b61d-08002795cb26
# ls
SUCCESS
```

说明部署正常，并且可以动态分配NFS的共享卷。

监控

Kubernetes 使得管理复杂环境变得更简单，但是对 kubernetes 本身的各种组件还有运行在 kubernetes 集群上的各种应用程序做到很好的洞察就很难了。Kubernetes 本身对应用程序的做了很多抽象，在生产环境下对这些不同的抽象组件的健康就是迫在眉睫的事情。

我们在安装 kubernetes 集群的时候，默认安装了 kubernetes 官方提供的 heapster 插件，可以对 kubernetes 集群上的应用进行简单的监控，获取 pod 级别的内存、**CPU**和网络监控信息，同时还能够通过 API 监控 kubernetes 中的基本资源监控指标。

然而，Prometheus 的出现让人眼前一亮，与 kubernetes 一样同样为 CNCF 中的项目，而且是第一个加入到 CNCF 中的项目。

Prometheus 是由 SoundCloud 开源监控告警解决方案，从 2012 年开始编写代码，再到 2015 年 GitHub 上开源以来，已经吸引了 9k+ 关注，以及很多大公司的使用；2016 年 Prometheus 成为继 k8s 后，第二名 CNCF(Cloud Native Computing Foundation) 成员。

作为新一代开源解决方案，很多理念与 Google SRE 运维之道不谋而合。

Heapster

Heapster作为kubernetes安装过程中默认安装的一个插件，见安装heapster插件。这对于集群监控十分有用，同时在Horizontal Pod Autoscaling中也用到了，HPA将Heapster作为 `Resource Metrics API` ，向其获取metric，做法是在 `kube-controller-manager` 中配置 `--api-server` 指向kube-aggregator，也可以使用heapster来实现，通过在启动heapster的时候指定 `--api-server=true` 。

Heapster可以收集Node节点上的cAdvisor数据，还可以按照kubernetes的资源类型来集合资源，比如Pod、Namespace域，可以分别获取它们的CPU、内存、网络和磁盘的metric。默认的metric数据聚合时间间隔是1分钟。

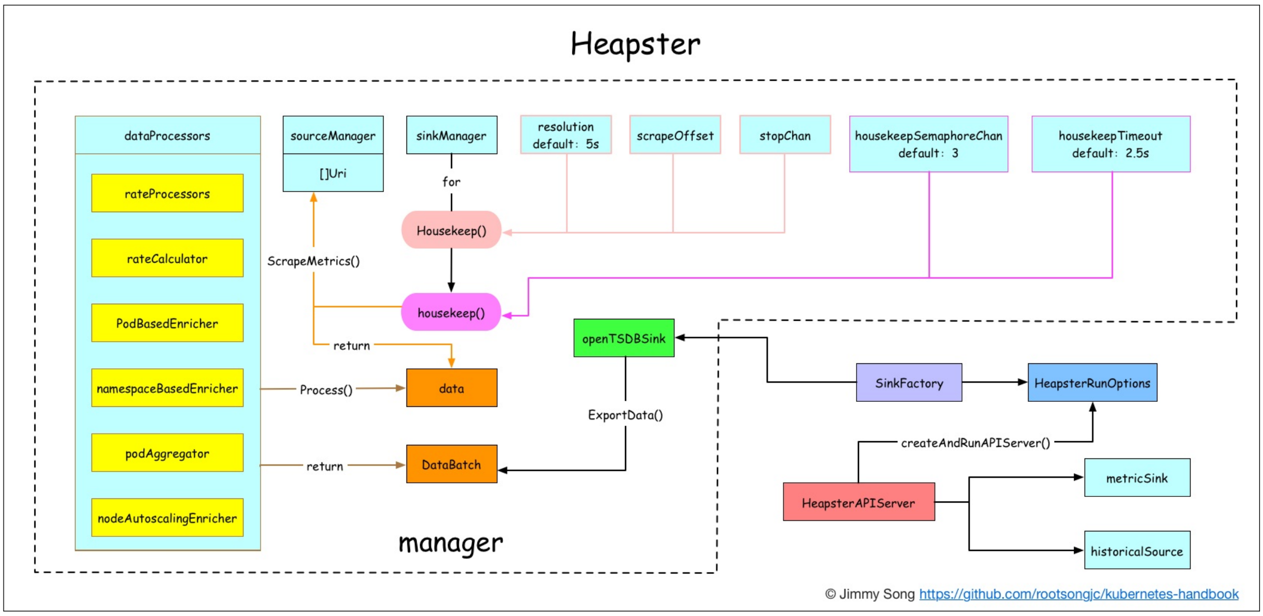
使用Heapster获取集群对象的metric数据

Heapster作为kubernetes安装过程中默认安装的一个插件，见安装heapster插件。这对于集群监控十分有用，同时在Horizontal Pod Autoscaling中也用到了，HPA将Heapster作为 `Resource Metrics API` ，向其获取metric，做法是在 `kube-controller-manager` 中配置 `--api-server` 指向kube-aggregator，也可以使用heapster来实现，通过在启动heapster的时候指定 `--api-server=true` 。

Heapster可以收集Node节点上的cAdvisor数据，还可以按照kubernetes的资源类型来集合资源，比如Pod、Namespace域，可以分别获取它们的CPU、内存、网络和磁盘的metric。默认的metric数据聚合时间间隔是1分钟。

架构

下面是Heapster架构图：



图片 - *Heapster*架构图

Heapser是用Go语言开发Kubernetes集群计算资源使用情况的数据采集工具，编译后可以直接以一个二进制文件运行，通过向heapster传递的参数来指定数据采集行为，这些数据可以选择多种sink方式，例如Graphite、influxDB、OpenTSDB、

ElasticSearch、Kafka等。

使用案例

Heapster使用起来很简单，本身就是二进制文件，直接使用命令行启动，也可以放在容器里运行，在作为kubernetes插件运行时，我们是直接放在容器中的，见安装heapster插件。

运行

下面是heapster的启动参数：

Flag	Description
--allowed-users string	comma-separated list of allowed users
--alsologtostderr	log to standard error as well as files
--api-server	Enable API server for the Metrics API. If set, the Metrics API will be served on --insecure-port (internally) and --secure-port (externally).
--authentication-kubeconfig string	kubeconfig file pointing at the 'core' kubernetes server with enough rights to create tokenaccessreviews.authentication.k8s.io.
--authentication-token-webhook-cache-ttl duration	The duration to cache responses from the webhook token authenticator. (default 10s)
--authorization-kubeconfig string	kubeconfig file pointing at the 'core' kubernetes server with enough rights to create subjectaccessreviews.authorization.k8s.io.
--authorization-webhook-cache-authorized-ttl duration	The duration to cache 'authorized' responses from the webhook authorizer. (default 10s)
--authorization-	

webhook-cache-unauthorized-ttl duration	The duration to cache 'unauthorized' responses from the webhook authorizer. (default 10s)
--bind-address ip	The IP address on which to listen for the --secure-port port. The associated interface(s) must be reachable by the rest of the cluster, and by CLI/web clients. If blank, all interfaces will be used (0.0.0.0). (default 0.0.0.0)
--cert-dir string	The directory where the TLS certs are located (by default /var/run/kubernetes). If --tls-cert-file and --tls-private-key-file are provided, this flag will be ignored. (default "/var/run/kubernetes")
--client-ca-file string	If set, any request presenting a client certificate signed by one of the authorities in the client-ca-file is authenticated with an identity corresponding to the CommonName of the client certificate.
--contention-profiling	Enable contention profiling. Requires --profiling to be set to work.
--disable-export	Disable exporting metrics in api/v1/metric-export
--enable-swagger-ui	Enables swagger ui on the apiserver at /swagger-ui
--heapster-port int	port used by the Heapster-specific APIs (default 8082)
--historical-source string	which source type to use for the historical API (should be exactly the same as one of the sink URIs), or empty to disable the historical API
--label-seperator string	seperator used for joining labels (default ",")
--listen-ip string	IP to listen on, defaults to all IPs
--log-backtrace-at traceLocation	when logging hits line file:N, emit a stack trace (default :0)
--log-dir string	If non-empty, write log files in this directory
--log-flush-frequency duration	Maximum number of seconds between log flushes (default 5s)

--logtostderr	log to standard error instead of files (default true)
--max-procs int	max number of CPUs that can be used simultaneously. Less than 1 for default (number of cores)
--metric-resolution duration	The resolution at which heapster will retain metrics. (default 1m0s)
--profiling	Enable profiling via web interface host:port/debug/pprof/ (default true)
--requestheader-allowed-names stringSlice	List of client certificate common names to allow to provide usernames in headers specified by --requestheader-username-headers. If empty, any client certificate validated by the authorities in --requestheader-client-ca-file is allowed.
--requestheader-client-ca-file string	Root certificate bundle to use to verify client certificates on incoming requests before trusting usernames in headers specified by --requestheader-username-headers
--requestheader-extra-headers-prefix stringSlice	List of request header prefixes to inspect. X-Remote-Extra- is suggested. (default [x-remote-extra-])
--requestheader-group-headers stringSlice	List of request headers to inspect for groups. X-Remote-Group is suggested. (default [x-remote-group])
--requestheader-username-headers stringSlice	List of request headers to inspect for usernames. X-Remote-User is common. (default [x-remote-user])
--secure-port int	The port on which to serve HTTPS with authentication and authorization. If 0, don't serve HTTPS at all. (default 6443)
--sink *flags.Uris	external sink(s) that receive data (default [])
--source *flags.Uris	source(s) to watch (default [])
--stderrthreshold severity	logs at or above this threshold go to stderr (default 2)

--tls-ca-file string	If set, this certificate authority will used for secure access from Admission Controllers. This must be a valid PEM-encoded CA bundle. Altneratively, the certificate authority can be appended to the certificate provided by --tls-cert-file.
--tls-cert string	file containing TLS certificate
--tls-cert-file string	File containing the default x509 Certificate for HTTPS. (CA cert, if any, concatenated after server cert). If HTTPS serving is enabled, and --tls-cert-file and --tls-private-key-file are not provided, a self-signed certificate and key are generated for the public address and saved to /var/run/kubernetes.
--tls-client-ca string	file containing TLS client CA for client cert validation
--tls-key string	file containing TLS key
--tls-private-key-file string	File containing the default x509 private key matching --tls-cert-file.
--tls-sni-cert-key namedCertKey	A pair of x509 certificate and private key file paths, optionally suffixed with a list of domain patterns which are fully qualified domain names, possibly with prefixed wildcard segments. If no domain patterns are provided, the names of the certificate are extracted. Non-wildcard matches trump over wildcard matches, explicit domain patterns trump over extracted names. For multiple key/certificate pairs, use the --tls-sni-cert-key multiple times. Examples: "example.key,example.crt" or "*.foo.com,foo.com:foo.key,foo.crt". (default [])
--v Level	log level for V logs
--version	print version info and exit
--vmodule moduleSpec	comma-separated list of pattern=N settings for file-filtered logging

Version

version: v1.4.0 commit: 546ab66f

API使用

Heapster提供RESTful API接口，下面以获取 `spark-cluster` namespace的memory usage为例讲解Heapster API的使用。

构造**URL**地址

https://172.20.0.113:6443/api/v1/proxy/namespaces/kube-system/services/heapster/api/v1/model/namespaces/spark-cluster/metrics/memory/usage?start=2017-10-16T09:14:00Z&end=2017-10-16T09:16:00Z

结果

访问该地址获取的结果是这样的：

```
{
  "metrics": [
   {
    "timestamp": "2017-10-16T09:14:00Z",
    "value": 322592768
   },
   {
    "timestamp": "2017-10-16T09:15:00Z",
    "value": 322592768
   },
   {
    "timestamp": "2017-10-16T09:16:00Z",
    "value": 322592768
   }
  ],
  "latestTimestamp": "2017-10-16T09:16:00Z"
 }
```

注意：Heapster中查询的所有值都是以最小单位为单位，比如CPU为1milicore，内存为B。

第一部分：**Heapster API**地址

`https://172.20.0.113:6443/api/v1/proxy/namespaces/kube-system/services/heapster/`

可以使用下面的命令获取:

```
$ kubectl cluster-info
Heapster is running at https://172.20.0.113:6443/api/v1/proxy/na
mespaces/kube-system/services/heapster
...
```

第二部分：**Heapster API**参数

`/api/v1/model/namespaces/spark-cluster/metrics/memory/usage`

表示查询的是 `spark-cluster` namespace中的 `memory/usage` 的metrics。

第三部分：时间片

`?start=2017-10-16T09:14:00Z&end=2017-10-16T09:16:00Z`

查询参数为时间片：包括start和end。使用 `RFC-3339` 时间格式，在Linux系统中可以这样获取：

```
$ date --rfc-3339="seconds"
2017-10-16 17:23:20+08:00
```

该时间中的空格替换成T，最后的 `+08:00` 替换成Z代表时区。可以只指定start时间，end时间自动设置为当前时间。

参考

- kubernetes metrics
- Heapster metric model
- Heapster storage schema

Prometheus

Prometheus 是由 SoundCloud 开源监控告警解决方案，从 2012 年开始编写代码，再到 2015 年 github 上开源以来，已经吸引了 9k+ 关注，以及很多大公司的使用；2016 年 Prometheus 成为继 k8s 后，第二名 CNCF(Cloud Native Computing Foundation) 成员。

作为新一代开源解决方案，很多理念与 Google SRE 运维之道不谋而合。

主要功能

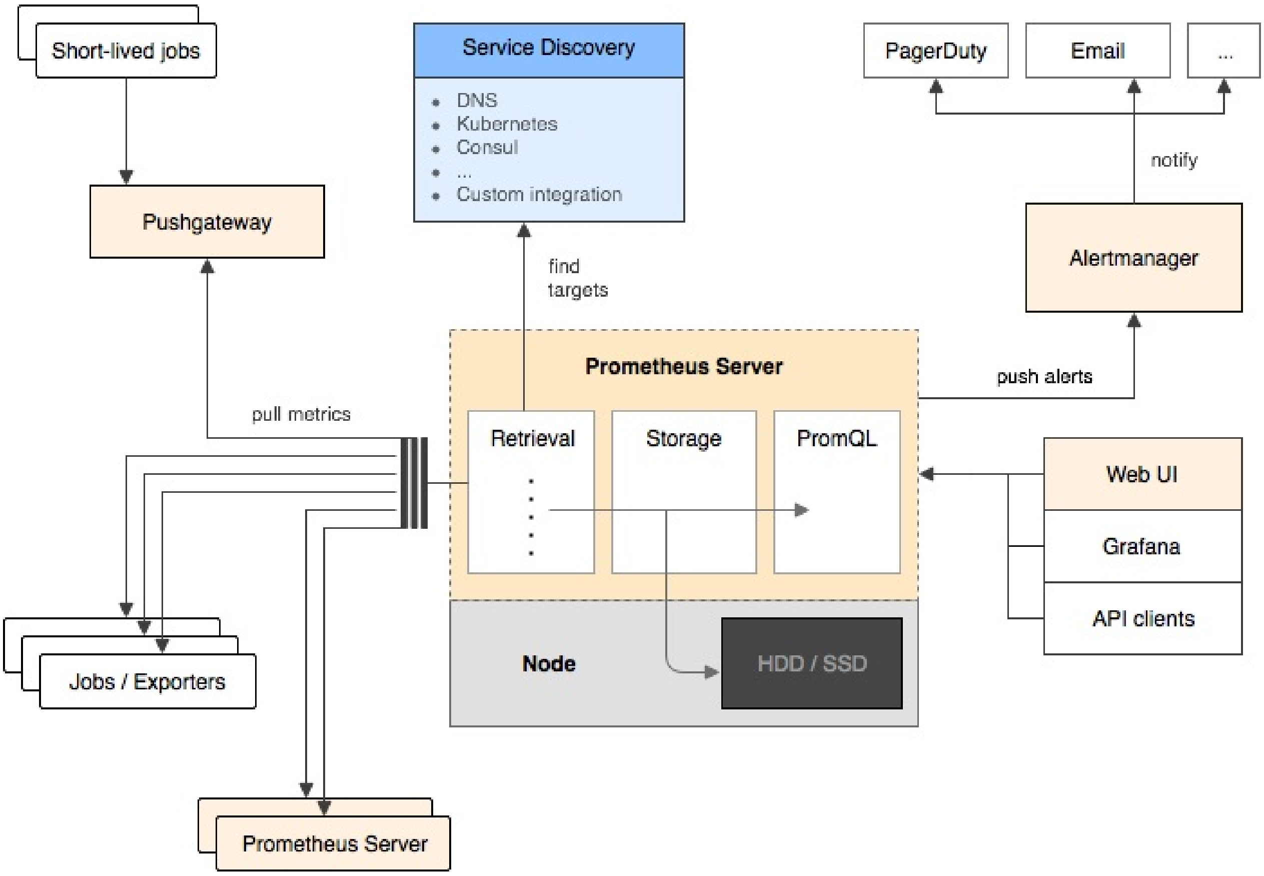
- 多维 数据模型（时序由 metric 名字和 k/v 的 labels 构成）。
- 灵活的查询语句（PromQL）。
- 无依赖存储，支持 local 和 remote 不同模型。
- 采用 http 协议，使用 pull 模式，拉取数据，简单易懂。
- 监控目标，可以采用服务发现或静态配置的方式。
- 支持多种统计数据模型，图形化友好。

核心组件

- Prometheus Server， 主要用于抓取数据和存储时序数据，另外还提供查询和 Alert Rule 配置管理。
- client libraries，用于对接 Prometheus Server, 可以查询和上报数据。
- push gateway ，用于批量，短期的监控数据的汇总节点，主要用于业务数据汇报等。
- 各种汇报数据的 exporters ，例如汇报机器数据的 node_exporter, 汇报 MongoDB 信息的 MongoDB exporter 等等。
- 用于告警通知管理的 alertmanager 。

基础架构

一图胜千言，先来张官方的架构图：



图片 - *Prometheus* 架构图

从这个架构图，也可以看出 Prometheus 的主要模块包含， Server, Exporters, Pushgateway, PromQL, Alertmanager, WebUI 等。

它大致使用逻辑是这样：

1. Prometheus server 定期从静态配置的 targets 或者服务发现的 targets 拉取数据。
2. 当新拉取的数据大于配置内存缓存区的时候，Prometheus 会将数据持久化到磁盘（如果使用 remote storage 将持久化到云端）。
3. Prometheus 可以配置 rules，然后定时查询数据，当条件触发的时候，会将 alert 推送到配置的 Alertmanager。
4. Alertmanager 收到警告的时候，可以根据配置，聚合，去重，降噪，最后发送警告。
5. 可以使用 API， Prometheus Console 或者 Grafana 查询和聚合数据。

注意

- Prometheus 的数据是基于时序的 float64 的值，如果你的数据值有更多类型，无法满足。
- Prometheus 不适合做审计计费，因为它的数据是按一定时间采集的，关注的更多是系统的运行瞬时状态以及趋势，即使有少量数据没有采集也能容忍，但是审计计费需要记录每个请求，并且数据长期存储，这个和 Prometheus 无法满足，可能需要采用专门的审计系统。

以上介绍来自 https://github.com/songjiayang/prometheus_practice/

使用Prometheus监控kubernetes集群

我们使用 Giantswarm 开源的 kubernetes-promethues 来监控 kubernetes 集群，所有的 YAML 文件可以在 ../manifests/prometheus 目录下找到。

需要用到的镜像有：

- harbor-001.jimmysong.io/library/prometheus-alertmanager:v0.7.1
- harbor-001.jimmysong.io/library/grafana:4.2.0
- harbor-001.jimmysong.io/library/giantswarm-tiny-tools:latest
- harbor-001.jimmysong.io/library/prom-prometheus:v1.7.0
- harbor-001.jimmysong.io/library/kube-state-metrics:v1.0.1
- harbor-001.jimmysong.io/library/dockermuenster-caddy:0.9.3
- harbor-001.jimmysong.io/library/prom-node-exporter:v0.14.0

同时备份到时速云：

- index.tenxcloud.com/jimmy/prometheus-alertmanager:v0.7.1
- index.tenxcloud.com/jimmy/grafana:4.2.0
- index.tenxcloud.com/jimmy/giantswarm-tiny-tools:latest
- index.tenxcloud.com/jimmy/prom-prometheus:v1.7.0
- index.tenxcloud.com/jimmy/kube-state-metrics:v1.0.1
- index.tenxcloud.com/jimmy/dockermuenster-caddy:0.9.3
- index.tenxcloud.com/jimmy/prom-node-exporter:v0.14.0

注：所有镜像都是从官方镜像仓库下载下。

部署

我将部署时需要用的的配置文件分成了 namespace、serviceaccount、configmaps、clusterrolebinding 和最后的部署 prometheus、grafana 的过程。

```
## 创建 monitoring namespaece
kubectl create -f prometheus-monitoring-ns.yaml
## 创建 serviceaccount
kubectl create -f prometheus-monitoring-serviceaccount.yaml
## 创建 configmaps
kubectl create -f prometheus-configmaps.yaml
## 创建 clusterrolebinding
kubectl create clusterrolebinding kube-state-metrics --clusterro
le=cluster-admin --serviceaccount=monitoring:kube-state-metrics
kubectl create clusterrolebinding prometheus --clusterrole=clust
er-admin --serviceaccount=monitoring:prometheus
## 部署 Prometheus
kubectl create -f prometheus-monitoring.yaml
```

访问 kubernetes 任何一个 node 上的 Grafana service 的 nodeport：



图片 - *Grafana*页面

该图中的数据显示明显有问题，还需要修正。

`prometheus-monitoring.yaml` 文件中有一个 Job 就是用来导入 grafana dashboard 配置信息的，如果该 Job 执行失败，可以单独在在 `monitoring` 的 namespace 中启动一个容器，将 `manifests/prometheus` 目录下的 json 文件复制到容器中，然后进入容器 json 文件的目录下执行：

```
 for file in *-datasource.json ; do
            if [ -e "$file" ] ; then
              echo "importing $file" &&
              curl --silent --fail --show-error \
                --request POST http://admin:admin@grafana:3000
/api/datasources \
                --header "Content-Type: application/json" \
                --data-binary "@$file" ;
              echo "" ;
            fi
          done ;
          for file in *-dashboard.json ; do
            if [ -e "$file" ] ; then
              echo "importing $file" &&
              ( echo '{"dashboard":'; \
                cat "$file"; \
                echo ',"overwrite":true,"inputs":[{"name":"DS_
PROMETHEUS","type":"datasource","pluginId":"prometheus","value":
"prometheus"}]}' ) \
              | jq -c '.' \
              | curl --silent --fail --show-error \
                --request POST http://admin:admin@grafana:3000
/api/dashboards/import \
                --header "Content-Type: application/json" \
                --data-binary "@-" ;
              echo "" ;
            fi
          done
```

这样也可以向 grafana 中导入 dashboard。

存在的问题

该项目的代码中存在几个问题。

1. RBAC 角色授权问题

需要用到两个 clusterrolebinding：

- `kube-state-metrics`，对应的 `serviceaccount` 是 `kube-state-metrics`
- `prometheus`，对应的 `serviceaccount` 是 `prometheus-k8s`

在部署 Prometheus 之前应该先创建 serviceaccount、clusterrole、clusterrolebinding 等对象，否则在安装过程中可能因为权限问题而导致各种错误，所以这些配置应该写在一个单独的文件中，而不应该跟其他部署写在一起，即使要写在一个文件中，也应该写在文件的最前面，因为使用 `kubectl` 部署的时候，kubectl 不会判断 YAML 文件中的资源依赖关系，只是简单的从头部开始执行部署，因此写在文件前面的对象会先部署。

解决方法

也可以绕过复杂的 RBAC 设置，直接使用下面的命令将对应的 serviceaccount 设置成 admin 权限，如下：

```
kubectl create clusterrolebinding kube-state-metrics --clusterro
le=cluster-admin --serviceaccount=monitoring:kube-state-metrics
kubectl create clusterrolebinding prometheus --clusterrole=clust
er-admin --serviceaccount=monitoring:prometheus
```

参考 RBAC——基于角色的访问控制

2. API 兼容问题

从 `kube-state-metrics` 日志中可以看出用户 kube-state-metrics 没有权限访问如下资源类型：

- *v1.Job
- *v1.PersistentVolumeClaim
- *v1beta1.StatefulSet
- *v2alpha1.CronJob

而在我们使用的 kubernetes 1.6.0 版本的集群中 API 路径跟 `kube-state-metrics` 中不同，无法 list 以上三种资源对象的资源。详情见：https://github.com/giantswarm/kubernetes-prometheus/issues/77

3. Job 中的权限认证问题

在 `grafana-import-dashboards` 这个 job 中有个 `init-containers` 其中指定的 command 执行错误，应该使用

```
curl -sX GET -H "Authorization:bearer `cat /var/run/secrets/kubernetes.io/serviceaccount/token`" -k https://kubernetes.default/api/v1/namespaces/monitoring/endpoints/grafana
```

不需要指定 csr 文件，只需要 token 即可。

参考 wait-for-endpoints init-containers fails to load with k8s 1.6.0 #56

参考

- Kubernetes Setup for Prometheus and Grafana
- RBAC——基于角色的访问控制
- wait-for-endpoints init-containers fails to load with k8s 1.6.0 #56

服务编排管理

Kubernetes虽然提供了多种容器编排对象，例如Deployment、StatefulSet、DeamonSet、Job等，还有多种基础资源封装例如ConfigMap、Secret、Serivce等，但是一个应用往往有多个服务，有的可能还要依赖持久化存储，当这些服务之间直接互相依赖，需要有一定的组合的情况下，使用YAML文件的方式配置应用往往十分繁琐还容易出错，这时候就需要服务编排工具。

服务编排管理工具就是构建在kubernetes的基础object之上，统筹各个服务之间的关系和依赖的。目前常用到的工具是 Helm。

使用Helm管理kubernetes应用

读完本文后您应该可以自己创建chart，并创建自己的私有chart仓库。

Helm是一个kubernetes应用的包管理工具，用来管理charts——预先配置好的安装包资源，有点类似于Ubuntu的APT和CentOS中的yum。

Helm chart是用来封装kubernetes原生应用程序的yaml文件，可以在你部署应用的时候自定义应用程序的一些metadata，便与应用程序的分发。

Helm和charts的主要作用：

- 应用程序封装
- 版本管理
- 依赖检查
- 便于应用程序分发

安装Helm

前提要求

- Kubernetes1.5以上版本
- 集群可访问到的镜像仓库
- 执行helm命令的主机可以访问到kubernetes集群

安装步骤

首先需要安装helm客户端

```
curl https://raw.githubusercontent.com/kubernetes/helm/master/scripts/get > get_helm.sh
chmod 700 get_helm.sh
./get_helm.sh
```

创建tiller的 `serviceaccount` 和 `clusterrolebinding`

```
kubectl create serviceaccount --namespace kube-system tiller
kubectl create clusterrolebinding tiller-cluster-rule --clusterr
ole=cluster-admin --serviceaccount=kube-system:tiller
```

然后安装helm服务端tiller

```
helm init -i jimmysong/kubernetes-helm-tiller:v2.8.1
```

(目前最新版v2.8.2，可以使用阿里云镜像，如： helm init --upgrade -i registry.cn-hangzhou.aliyuncs.com/google_containers/tiller:v2.8.2 --stable-repo-url https://kubernetes.oss-cn-hangzhou.aliyuncs.com/charts）

我们使用 `-i` 指定自己的镜像，因为官方的镜像因为某些原因无法拉取，官方镜像地址是： `gcr.io/kubernetes-helm/tiller:v2.8.1` ，我在DockerHub上放了一个备份 `jimmysong/kubernetes-helm-tiller:v2.8.1` ，该镜像的版本与helm客户端的版本相同，使用 `helm version` 可查看helm客户端版本。

为应用程序设置 `serviceAccount` ：

```
kubectl patch deploy --namespace kube-system tiller-deploy -p '{
"spec":{"template":{"spec":{"serviceAccount":"tiller"}}}}'
```

检查是否安装成功：

```
$ kubectl -n kube-system get pods|grep tiller
tiller-deploy-2372561459-f6p0z         1/1       Running   0
      1h
$ helm version
Client: &version.Version{SemVer:"v2.3.1", GitCommit:"32562a3040b
b5ca690339b9840b6f60f8ce25da4", GitTreeState:"clean"}
Server: &version.Version{SemVer:"v2.3.1", GitCommit:"32562a3040b
b5ca690339b9840b6f60f8ce25da4", GitTreeState:"clean"}
```

创建自己的chart

我们创建一个名为 `mychart` 的chart，看一看chart的文件结构。

```
$ helm create mongodb
$ tree mongodb
mongodb
├── Chart.yaml #Chart本身的版本和配置信息
├── charts #依赖的chart
├── templates #配置模板目录
│   ├── NOTES.txt #helm提示信息
│   ├── _helpers.tpl #用于修改kubernetes objcet配置的模板
│   ├── deployment.yaml #kubernetes Deployment object
│   └── service.yaml #kubernetes Serivce
└── values.yaml #kubernetes object configuration

2 directories, 6 files
```

模板

`Templates` 目录下是yaml文件的模板，遵循Go template语法。使用过Hugo的静态网站生成工具的人应该对此很熟悉。

我们查看下 `deployment.yaml` 文件的内容。

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: {{ template "fullname" . }}
  labels:
    chart: "{{ .Chart.Name }}-{{ .Chart.Version | replace "+" "_
" }}"
spec:
  replicas: {{ .Values.replicaCount }}
  template:
    metadata:
      labels:
        app: {{ template "fullname" . }}
    spec:
      containers:
      - name: {{ .Chart.Name }}
        image: "{{ .Values.image.repository }}:{{ .Values.image.
tag }}"
        imagePullPolicy: {{ .Values.image.pullPolicy }}
        ports:
        - containerPort: {{ .Values.service.internalPort }}
        livenessProbe:
          httpGet:
            path: /
            port: {{ .Values.service.internalPort }}
        readinessProbe:
          httpGet:
            path: /
            port: {{ .Values.service.internalPort }}
        resources:
{{ toyaml .Values.resources | indent 12 }}
```

这是该应用的Deployment的yaml配置文件，其中的双大括号包扩起来的部分是Go template，其中的Values是在 `values.yaml` 文件中定义的：

```yaml
# Default values for mychart.
# This is a yaml-formatted file.
# Declare variables to be passed into your templates.
replicaCount: 1
image:
  repository: nginx
  tag: stable
  pullPolicy: IfNotPresent
service:
  name: nginx
  type: ClusterIP
  externalPort: 80
  internalPort: 80
resources:
  limits:
    cpu: 100m
    memory: 128Mi
  requests:
    cpu: 100m
    memory: 128Mi
```

比如在 `Deployment.yaml` 中定义的容器镜像 `image: "{{ .Values.image.repository }}:{{ .Values.image.tag }}"` 其中的：

- `.Values.image.repository` 就是nginx
- `.Values.image.tag` 就是stable

以上两个变量值是在create chart的时候自动生成的默认值。

我们将默认的镜像地址和tag改成我们自己的镜像 `harbor-001.jimmysong.io/library/nginx:1.9` 。

检查配置和模板是否有效

当使用kubernetes部署应用的时候实际上讲templates渲染成最终的kubernetes能够识别的yaml格式。

使用 `helm install --dry-run --debug <chart_dir>` 命令来验证chart配置。该输出中包含了模板的变量配置与最终渲染的yaml文件。

```
$ helm install --dry-run --debug mychart
Created tunnel using local port: '58406'
SERVER: "localhost:58406"
CHART PATH: /Users/jimmy/Workspace/github/bitnami/charts/incubat
or/mean/charts/mychart
```

```
NAME:   filled-seahorse
REVISION: 1
RELEASED: Tue Oct 24 18:57:13 2017
CHART: mychart-0.1.0
USER-SUPPLIED VALUES:
{}

COMPUTED VALUES:
image:
  pullPolicy: IfNotPresent
  repository: harbor-001.jimmysong.io/library/nginx
  tag: 1.9
replicaCount: 1
resources:
  limits:
    cpu: 100m
    memory: 128Mi
  requests:
    cpu: 100m
    memory: 128Mi
service:
  externalPort: 80
  internalPort: 80
  name: nginx
  type: ClusterIP

HOOKS:
MANIFEST:

---
# Source: mychart/templates/service.yaml
apiVersion: v1
kind: Service
metadata:
  name: filled-seahorse-mychart
  labels:
    chart: "mychart-0.1.0"
spec:
  type: ClusterIP
  ports:
  - port: 80
    targetPort: 80
    protocol: TCP
    name: nginx
  selector:
    app: filled-seahorse-mychart

---
# Source: mychart/templates/deployment.yaml
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: filled-seahorse-mychart
```

```
    labels:
      chart: "mychart-0.1.0"
  spec:
    replicas: 1
    template:
      metadata:
        labels:
          app: filled-seahorse-mychart
      spec:
        containers:
        - name: mychart
          image: "harbor-001.jimmysong.io/library/nginx:1.9"
          imagePullPolicy: IfNotPresent
          ports:
          - containerPort: 80
          livenessProbe:
            httpGet:
              path: /
              port: 80
          readinessProbe:
            httpGet:
              path: /
              port: 80
          resources:
              limits:
                cpu: 100m
                memory: 128Mi
              requests:
                cpu: 100m
                memory: 128Mi
```

我们可以看到Deployment和Service的名字前半截由两个随机的单词组成，最后才是我们在 `values.yaml` 中配置的值。

部署到kubernetes

在 `mychart` 目录下执行下面的命令将nginx部署到kubernetes集群上。

```
helm install .
NAME:   eating-hound
LAST DEPLOYED: Wed Oct 25 14:58:15 2017
NAMESPACE: default
STATUS: DEPLOYED

RESOURCES:
==> v1/Service
NAME                  CLUSTER-IP     EXTERNAL-IP  PORT(S)  AGE
eating-hound-mychart  10.254.135.68  <none>       80/TCP   0s

==> extensions/v1beta1/Deployment
NAME                  DESIRED  CURRENT  UP-TO-DATE  AVAILABLE  A
GE
eating-hound-mychart  1        1        1           0          0
s


NOTES:
1. Get the application URL by running these commands:
  export POD_NAME=$(kubectl get pods --namespace default -l "app
=eating-hound-mychart" -o jsonpath="{.items[0].metadata.name}")
  echo "Visit http://127.0.0.1:8080 to use your application"
  kubectl port-forward $POD_NAME 8080:80
```

现在nginx已经部署到kubernetes集群上，本地执行提示中的命令在本地主机上访问到nginx实例。

```
export POD_NAME=$(kubectl get pods --namespace default -l "app=e
ating-hound-mychart" -o jsonpath="{.items[0].metadata.name}")
echo "Visit http://127.0.0.1:8080 to use your application"
kubectl port-forward $POD_NAME 8080:80
```

在本地访问 `http://127.0.0.1:8080` 即可访问到nginx。

查看部署的relaese

```
$ helm list
NAME            REVISION    UPDATED                     STATUS
    CHART           NAMESPACE
eating-hound    1           Wed Oct 25 14:58:15 2017    DEPLOYED
    mychart-0.1.0   default
```

删除部署的release

```
$ helm delete eating-hound
release "eating-hound" deleted
```

打包分享

我们可以修改 `Chart.yaml` 中的helm chart配置信息，然后使用下列命令将chart打包成一个压缩文件。

```
helm package .
```

打包出 `mychart-0.1.0.tgz` 文件。

依赖

我们可以在 `requirement.yaml` 中定义应用所依赖的chart，例如定义对 `mariadb` 的依赖：

```
dependencies:
- name: mariadb
  version: 0.6.0
  repository: https://kubernetes-charts.storage.googleapis.com
```

使用 `helm lint .` 命令可以检查依赖和模板配置是否正确。

安装源

#

使用第三方chat库

- 添加fabric8库

```
$helm repo add fabric8 https://fabric8.io/helm
```

- 搜索fabric8提供的工具（主要就是fabric8-platform工具包，包含了CI,CD的全套工具）

```
$helm search fabric8
```

#

我们在前面安装chart可以通过HTTP server的方式提供。

```
$ helm serve
Regenerating index. This may take a moment.
Now serving you on 127.0.0.1:8879
```

访问 `http://localhost:8879` 可以看到刚刚安装的chart。



图片 - *Helm chart*源

点击链接即可以下载chart的压缩包。

注意事项

下面列举一些常见问题，和在解决这些问题时候的注意事项。

服务依赖管理

所有使用helm部署的应用中如果没有特别指定chart的名字都会生成一个随机的 `Release name` ，例如 `romping-frog` 、 `sexy-newton` 等，跟启动docker容器时候容器名字的命名规则相同，而真正的资源对象的名字是在YAML文件中定义

的名字，我们成为 `App name` ，两者连接起来才是资源对象的实际名字： `Release name` - `App name` 。

而使用helm chart部署的包含依赖关系的应用，都会使用同一套 `Release name` ，在配置YAML文件的时候一定要注意在做服务发现时需要配置的服务地址，如果使用环境变量的话，需要像下面这样配置。

```
env:
 - name: SERVICE_NAME
   value: "{{ .Release.Name }}-{{ .Values.image.env.SERVICE_NAME }}"
```

这是使用了Go template的语法。至于 `{{ .Values.image.env.SERVICE_NAME }}` 的值是从 `values.yaml` 文件中获取的，所以需要在 `values.yaml` 中增加如下配置：

```
image:
  env:
    SERVICE_NAME: k8s-app-monitor-test
```

解决本地chart依赖

在本地当前chart配置的目录下启动helm server，我们不指定任何参数，直接使用默认端口启动。

```
helm serve
```

将该repo加入到repo list中。

```
helm repo add local http://localhost:8879
```

在浏览器中访问http://localhost:8879可以看到所有本地的chart。

然后下载依赖到本地。

```
helm dependency update
```

这样所有的chart都会下载到本地的 `charts` 目录下。

设置helm命令自动补全

为了方便helm命令的使用，helm提供了自动补全功能，如果使用zsh请执行：

```
source <(helm completion zsh)
```

如果使用bash请执行：

```
source <(helm completion bash)
```

部署MEAN测试案例

MEAN是用来构建网站和web应用的免费开源的JavaScript软件栈，该软件栈包括MongoDB、Express.js、Angular和Node.js。

下载**charts**

```
$ git clone https://github.com/bitnami/charts.git
$ cd charts/incubator/mean
$ helm dep list
NAME        VERSION     REPOSITORY
            STATUS
mongodb     0.4.x       https://kubernetes-charts.storage.googleap
is.com/     missing
```

缺少mongodb的依赖，需要更新一下chart。

注：`https://kubernetes-charts.storage.googleapis.com/` 是Google维护的chart库，访问该地址可以看到所有的chart列表。

```
$ helm dep update
Hang tight while we grab the latest from your chart repositories
...
...Unable to get an update from the "local" chart repository (ht
tp://127.0.0.1:8879/charts):
    Get http://127.0.0.1:8879/charts/index.yaml: dial tcp 127.0.
0.1:8879: getsockopt: connection refused
...Successfully got an update from the "stable" chart repository
Update Complete. ⎈Happy Helming!⎈
Saving 1 charts
Downloading mongodb from repo https://kubernetes-charts.storage.
googleapis.com/
```

所有的image都在 `values.yaml` 文件中配置。

下载缺失的chart。

```
$ helm dep build
Hang tight while we grab the latest from your chart repositories
...
...Unable to get an update from the "local" chart repository (ht
tp://127.0.0.1:8879/charts):
    Get http://127.0.0.1:8879/charts/index.yaml: dial tcp 127.0.
0.1:8879: getsockopt: connection refused
...Successfully got an update from the "stable" chart repository
Update Complete. ⎈Happy Helming!⎈
Saving 1 charts
Downloading mongodb from repo https://kubernetes-charts.storage.
googleapis.com/
```

修改mongodb chart配置

将刚才下载的 `charts/mongodb-0.4.17.tgz` 给解压后，修改其中的配置：

- 将 `persistence` 下的 `enabled` 设置为false
- 将image修改为我们的私有镜像：harbor-001.jimmysong.io/library/bitnami-mongodb:3.4.9-r1

执行 `helm install --dry-run --debug .` 确定模板无误。

将修改后的mongodb chart打包，在mongodb的目录下执行：

```
helm package .
```

现在再访问前面启动的helm server `http://localhost:8879` 将可以在页面上看到mongodb-0.4.17这个chart。

我们对官方chart配置做了如下修改后推送到了自己的chart仓库：

- `requirements.yaml` 和 `requirements.lock` 文件中的 `repository` 为 `http://localhost:8879`
- 将 `values.yaml` 中的 `storageClass` 设置为 `null`
- 将 `values.yaml` 中的 `Image` 都改为私有镜像
- `repositroy` 都设置为 `http://localhost:8879`

注：因为我们没有使用PVC所以将所有的关于持久化存储的配置都设置为false了。

部署MEAN

在 `mean` 目录下执行：

```
helm install .
NAME:   orbiting-platypus
LAST DEPLOYED: Wed Oct 25 16:21:48 2017
NAMESPACE: default
STATUS: DEPLOYED

RESOURCES:
==> v1/Secret
NAME                       TYPE    DATA  AGE
orbiting-platypus-mongodb  Opaque  2     2s

==> v1/ConfigMap
NAME                    DATA  AGE
orbiting-platypus-mean  1     2s

==> v1/Service
NAME                       CLUSTER-IP      EXTERNAL-IP  PORT(S)
   AGE
orbiting-platypus-mongodb  10.254.144.208  <none>       27017/TC
P  2s
orbiting-platypus-mean     10.254.165.23   <none>       80/TCP
   2s

==> extensions/v1beta1/Deployment
NAME                       DESIRED  CURRENT  UP-TO-DATE  AVAILAB
LE  AGE
orbiting-platypus-mean     1        1        1           0
    2s
orbiting-platypus-mongodb  1        1        1           0
    2s


NOTES:

Get the URL of your Node app  by running:

  export POD_NAME=$(kubectl get pods --namespace default -l "app
=orbiting-platypus-mean" -o jsonpath="{.items[0].metadata.name}")

  echo http://127.0.0.1:8080/
  kubectl port-forward $POD_NAME 8080:80
```

这样MEAN软件栈就部署到你的kuberentes集群里面了（默认是在default namespace下）。

验证检查

为了验证MEAN是否安装成功过，可以使用 `kubectl get pods` 查看pod是否启动完成，会先启动mongodb的pod，然后启动MEAN中的4步init。

访问Web UI

在Ingress中增加如下配置：

```yaml
  - host: mean.jimmysong.io
    http:
      paths:
      - backend:
          serviceName: orbiting-platypus-mean
          servicePort: 80
        path: /
```

然后在页面中更新ingress:

```
kubectl repalce -f ingress.yaml
```

关于Ingress配置请参考：边缘节点配置

然后在本地的 `/etc/hosts` 文件中增加一条配置：

```
172.20.0.119 mean.jimmysong.io
```

注：172.20.0.119即边缘节点的VIP。

因为该页面需要加载google的angularjs、还有两个css在国内无法访问，可以使用curl测试：

```
curl mean.jimmysong.io
```

将会返回HTML内容：

```html
<!doctype html>

<!-- ASSIGN OUR ANGULAR MODULE -->
<html ng-app="scotchTodo">

<head>
    <!-- META -->
    <meta charset="utf-8">
```

```
    <meta name="viewport" content="width=device-width, initial-s
cale=1">
    <!-- Optimize mobile viewport -->

    <title>Node/Angular Todo App</title>

    <!-- SCROLLS -->
    <link rel="stylesheet" href="//netdna.bootstrapcdn.com/boots
trap/3.0.0/css/bootstrap.min.css">
    <!-- load bootstrap -->
    <link rel="stylesheet" href="//netdna.bootstrapcdn.com/font-
awesome/4.0.3/css/font-awesome.min.css">
    <style>
        html {
            overflow-y: scroll;
        }

        body {
            padding-top: 50px;
        }

        #todo-list {
            margin-bottom: 30px;
        }

        #todo-form {
            margin-bottom: 50px;
        }
    </style>

    <!-- SPELLS -->
    <script src="//ajax.googleapis.com/ajax/libs/angularjs/1.2.1
6/angular.min.js"></script>
    <!-- load angular -->

    <script src="js/controllers/main.js"></script>
    <!-- load up our controller -->
    <script src="js/services/todos.js"></script>
    <!-- load our todo service -->
    <script src="js/core.js"></script>
    <!-- load our main application -->

</head>
<!-- SET THE CONTROLLER -->

<body ng-controller="mainController">
    <div class="container">

        <!-- HEADER AND TODO COUNT -->
        <div class="jumbotron text-center">
            <h1>I'm a Todo-aholic <span class="label label-info">
{{ todos.length }}</span></h1>
        </div>
```

```
        <!-- TODO LIST -->
        <div id="todo-list" class="row">
            <div class="col-sm-4 col-sm-offset-4">



                <!-- LOOP OVER THE TODOS IN $scope.todos -->
                <div class="checkbox" ng-repeat="todo in todos">
                    <label>
                        <input type="checkbox" ng-click="deleteT
odo(todo._id)"> {{ todo.text }}
                    </label>
                </div>

                <p class="text-center" ng-show="loading">
                    <span class="fa fa-spinner fa-spin fa-3x"></
span>
                </p>

            </div>
        </div>

        <!-- FORM TO CREATE TODOS -->
        <div id="todo-form" class="row">
            <div class="col-sm-8 col-sm-offset-2 text-center">
                <form>
                    <div class="form-group">

                        <!-- BIND THIS VALUE TO formData.text IN
 ANGULAR -->
                        <input type="text" class="form-control i
nput-lg text-center" placeholder="I want to buy a puppy that wil
l love me forever" ng-model="formData.text">
                    </div>

                    <!-- createToDo() WILL CREATE NEW TODOS -->
                    <button type="submit" class="btn btn-primary
 btn-lg" ng-click="createTodo()">Add</button>
                </form>
            </div>
        </div>

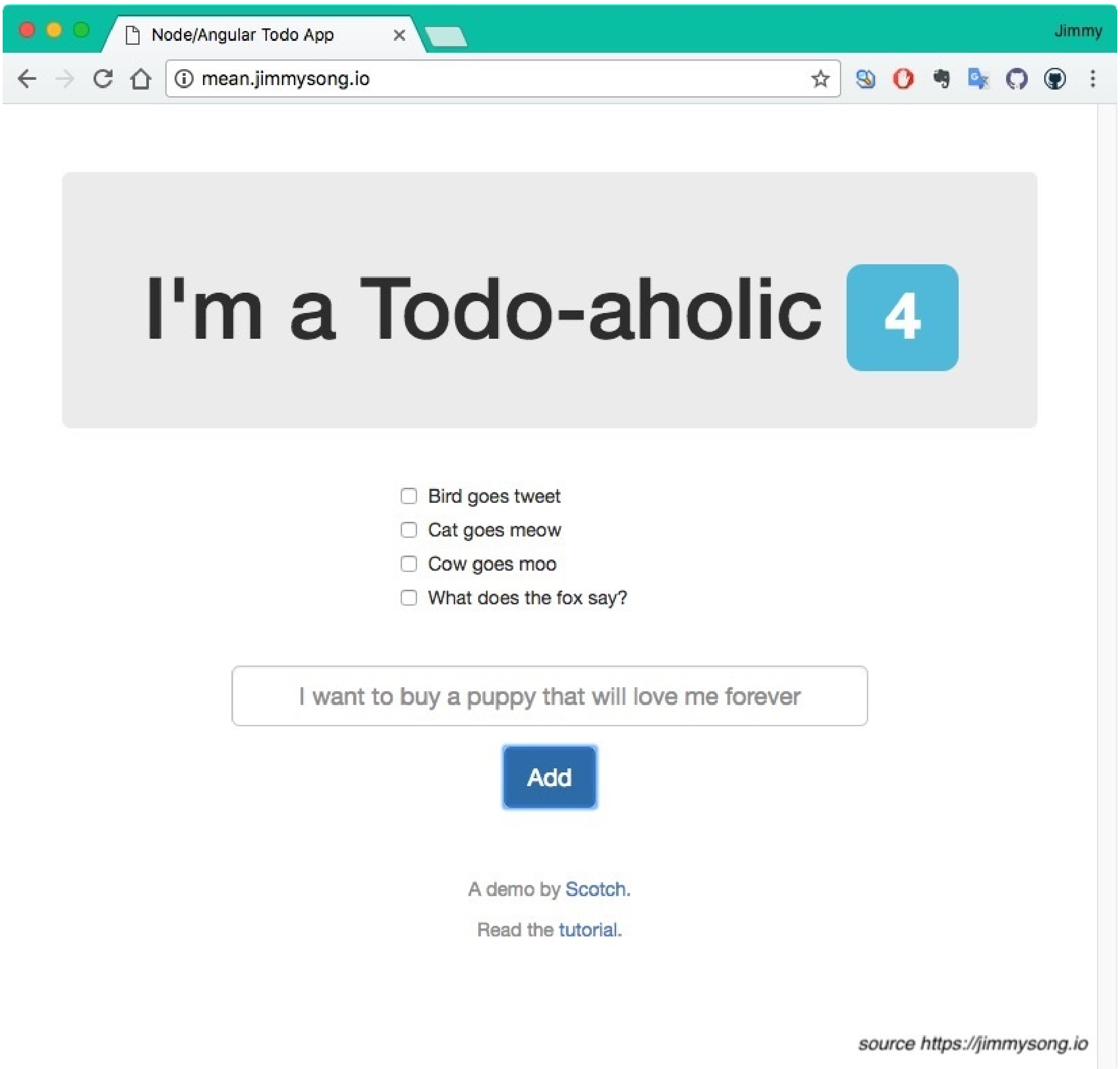
        <div class="text-center text-muted">
            <p>A demo by <a href="http://scotch.io">Scotch</a>.</
p>
            <p>Read the <a href="http://scotch.io/tutorials/java
script/creating-a-single-page-todo-app-with-node-and-angular">tu
torial</a>.</p>
        </div>

    </div>
```

```
</body>

</html>
```

访问 http://mean.jimmysong.io 可以看到如下界面，我在其中添加几条todo：



图片 - *TODO*应用的*Web*页面

注：Todo中的文字来自*What does the fox say*？

测试完成后可以使用下面的命令将mean chart推送的本地chart仓库中。

在mean目录下执行：

```
helm package .
```

再次刷新 `http://localhost:8879` 将可以看到如下三个chart：

- mean
  - mean-0.1.3
- mongodb
  - mongodb-0.4.17
- mychart
  - mychart-0.1.0

参考

- Deploy, Scale And Upgrade An Application On Kubernetes With Helm
- Helm charts
- Go template
- Helm docs
- How To Create Your First Helm Chart
- Speed deployment on Kubernetes with Helm Chart – Quick yaml example from scratch

构建私有Chart仓库

使用Chart便于封装和管理kubernetes中的应用，因此当企业内部的应用多了以后，互相依赖、部署环境复杂之后，原先的直接使用yaml文件的管理方式已经不再适应生产的需要，因此我们有必要构建自己的chart仓库。本文中我们将使用 GitHub Pages 来构建我们自己的 chart 仓库。

目的

我们需要构建一个GitHub pages存储所有chart的压缩文件，最好还要有一个前端来展示和搜索chart。

什么是Chart

Chart是helm管理的应用的打包格式。它包括如下特征：

- Chart中包括一系列的yaml格式的描述文件。
- 一个Chart只用来部署单个的应用的，不应该过于复杂，不应该包含多个依赖，相当于一个微服务。

Chart有特定的目录结构，可以打包起来进行版本控制。

Chart的组成结构

我们以nginx的chart为例，讲解chart的组成结构。

```
nginx/
  Chart.yaml          # 必须：一个包含chart的名称、版本和启用条件信息等的YAML文件
  LICENSE             # 可选: chart的许可证
  README.md           # 可选: 使用说明
  requirements.yaml   # 可选: 该chart的依赖配置
  values.yaml         # 必须：该chart的默认配置值
  charts/             # 可选: 包含该chart依赖的chart
  templates/          # 可选：kubernetes manifest文件模板，用于生成kubernetes yaml文件
  templates/NOTES.txt # 可选: 该chart的使用说明和提示信息文本文件，作为helm install后的提示信息
```

Chart的安装方式

安装chart主要分为安装本地定义的chart和远程chart仓库中的chart两种方式。

安装本地**chart**

- 指定本地chart目录：helm install .
- 指定本地chart压缩包：helm install nginx-1.2.3.tgz

安装**chart**仓库中的**chart**

- 使用默认的远程仓库：helm install stable/nginx
- 使用指定的仓库：helm install localhost:8879/nginx-1.2.3.tgz

实际上可以将chart打包后作为静态文件托管到web服务器上，例如GitHub pages作为chart仓库也可以。

依赖管理

有两种方式来管理chart的依赖。

- 直接在本的chart的 `charts` 目录下定义
- 通过在 `requirements.yaml` 文件中定义依赖的chart

在每个chart的 `charts` 目录下可以定义依赖的子chart。子chart有如下特点：

- 无法访问父chart中的配置
- 父chart可以覆盖子chart中的配置

Chart仓库

Chart 仓库（repository）是一个用来托管 `index.yaml` 文件和打包好的chart文件的web服务器。当前chart仓库本身没有设置身份和权限验证，查看此链接获取该问题的最新进展。

因为chart仓库只是一个HTTP服务，通过HTTP GET获取YAML文件和chart的压缩包，所以可以将这些文件存储在web服务器中，例如GCS、Amazon S3、GitHub Pages等。

关于chart仓库的更多信息请参考Helm chart文档。

使用GitHub pages托管charts

我们在上文中说到，chart可以使用GitHub pages做存储，接下来我们将会把之前够够构建的chart上传到GitHub pages并在helm中新增一个repo。

构建Monocular UI

参考 Monocular UI 构建UI。

克隆项目到本地

```
git clone https://github.com/kubernetes-helm/monocular.git
```

依赖环境

- Angular 2
- angular/cli
- Typescript
- Sass
- Webpack
- Bootstrap

在 `monoclar/scr/ui` 目录下执行以下命令安装依赖：

```
yarn install
npm install -g @angular/cli
npm install -g typescript
npm install -g webpack
```

运行

使用docker-compose

最简单的运行方式使用docker-compose：

```
docker-comopse up
```

该命令需要用到如下镜像：

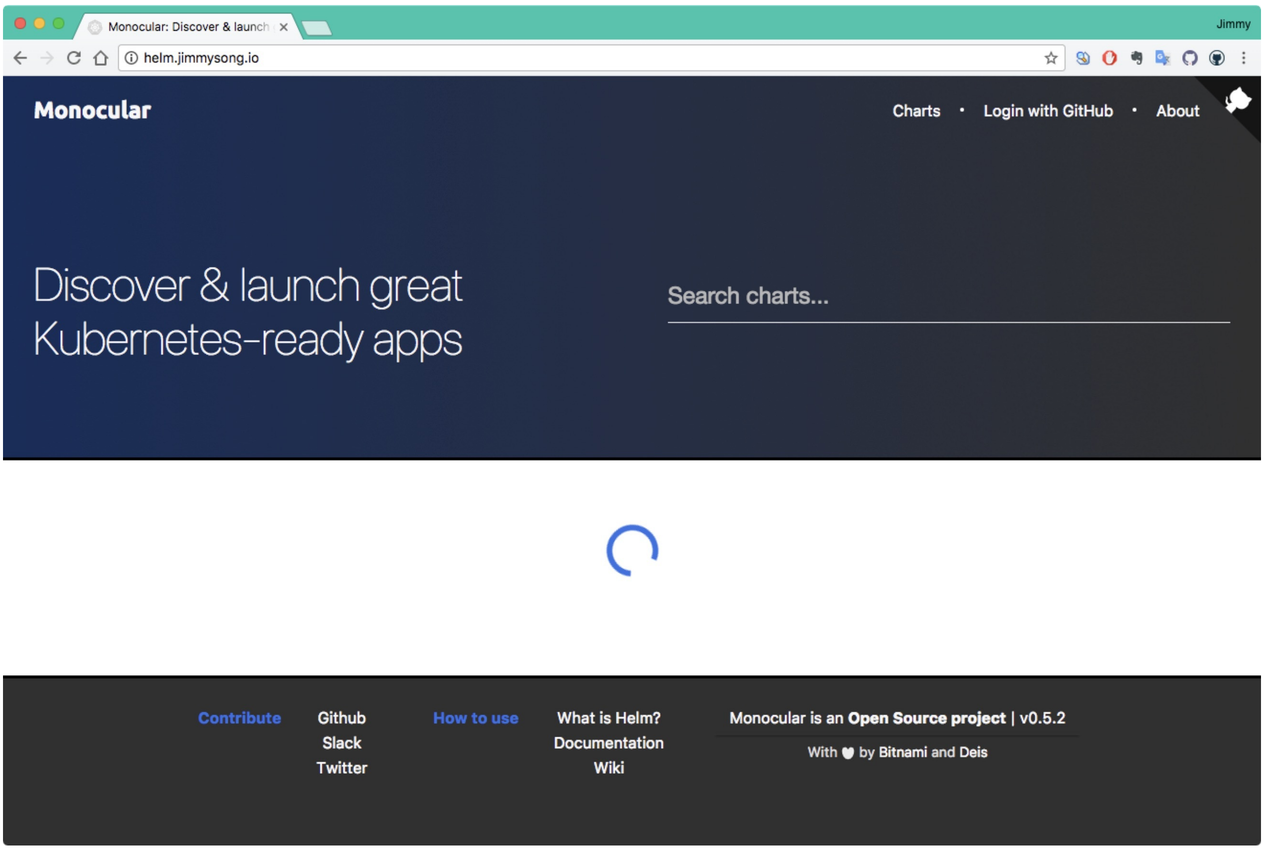
- bitnami/mongodb:3
- bitnami/node:8
- quay.io/deis/go-dev:v1.5.0

会有一个很长的build过程，构建失败。

使用helm

首先需要已在本地安装了helm，并在kubernetes集群中安装了tiller，见使用helm管理kubernetes应用。

```
# 需要安装nginx ingress
$ helm install stable/nginx-ingress
$ helm repo add monocular https://kubernetes-helm.github.io/mono
cular
$ helm install monocular/monocular
```



图片 - *Helm monocular*界面

因为nginx ingress配置问题，官方的chart中api与ui使用的是同样的domain name，我使用的是traefik ingress， `api` 访问不到，所以加载不了chart。

参考

- Monocular UI
- Helm Chart - GitHub
- 简化Kubernetes应用部署工具-Helm之应用部署
- Speed deployment on Kubernetes with Helm Chart – Quick YAML example from scratch
- Using a private github repo as helm chart repo (https access)

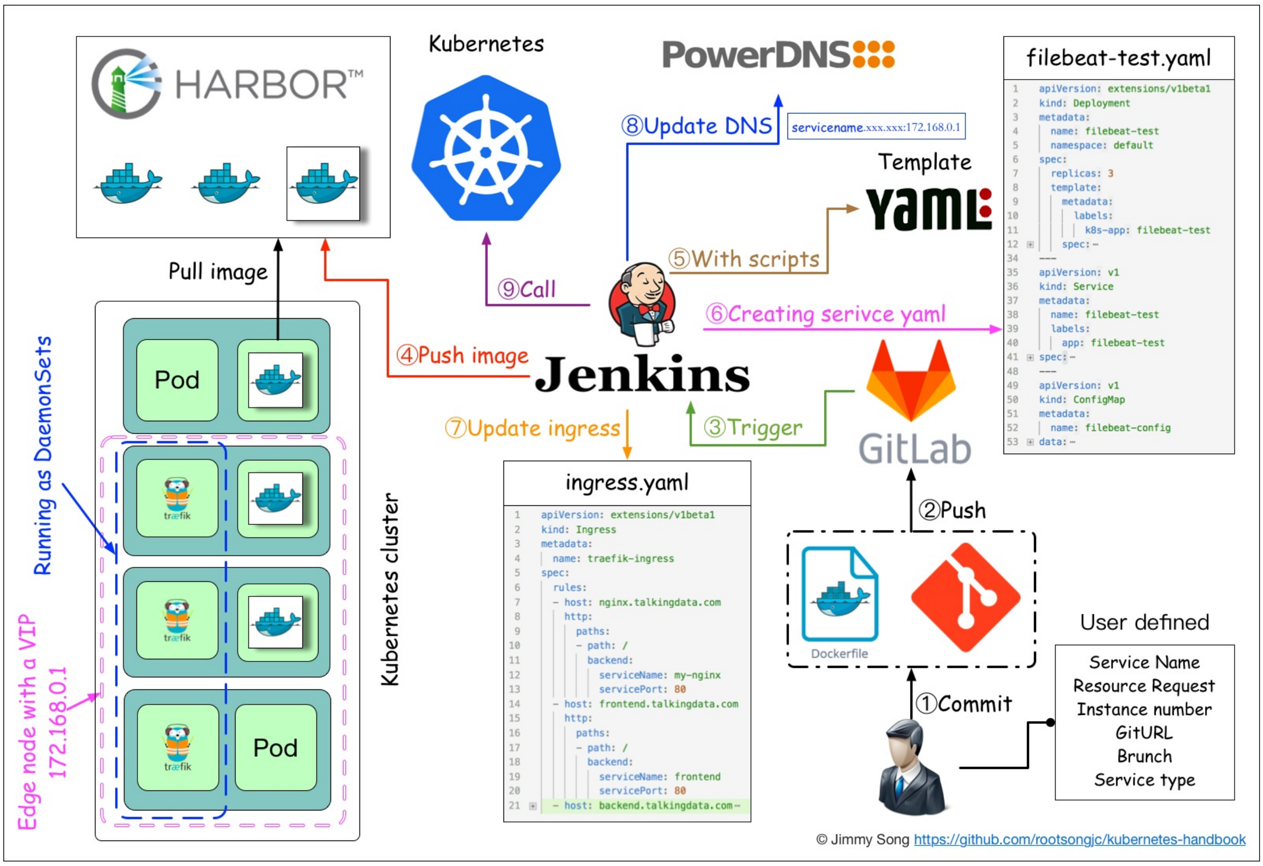
Updated at 2018-05-09 14:15:47

持续集成与发布

持续集成与发布，简称CI/CD，是微服务构建的重要环节，也是DevOps中推崇的方法论。如何在kubernetes中使用持续构建与发布工具？可以既可以与企业内部原有的持续构建集成，例如Jenkins，也可以在kubernetes中部署一套新的持续构建与发布工具，例如Drone。

使用Jenkins进行持续集成与发布

我们基于Jenkins的CI/CD流程如下所示。



图片 - 基于*Jenkins*的持续集成与发布

流程说明

应用构建和发布流程说明。

1. 用户向Gitlab提交代码，代码中必须包含 `Dockerfile`
2. 将代码提交到远程仓库
3. 用户在发布应用时需要填写git仓库地址和分支、服务类型、服务名称、资源数量、实例个数，确定后触发Jenkins自动构建
4. Jenkins的CI流水线自动编译代码并打包成docker镜像推送到Harbor镜像仓库
5. Jenkins的CI流水线中包括了自定义脚本，根据我们已准备好的kubernetes的YAML模板，将其中的变量替换成用户输入的选项

6. 生成应用的kubernetes YAML配置文件
7. 更新Ingress的配置，根据新部署的应用的名称，在ingress的配置文件中增加一条路由信息
8. 更新PowerDNS，向其中插入一条DNS记录，IP地址是边缘节点的IP地址。关于边缘节点，请查看边缘节点配置
9. Jenkins调用kubernetes的API，部署应用

使用Drone进行持续构建与发布

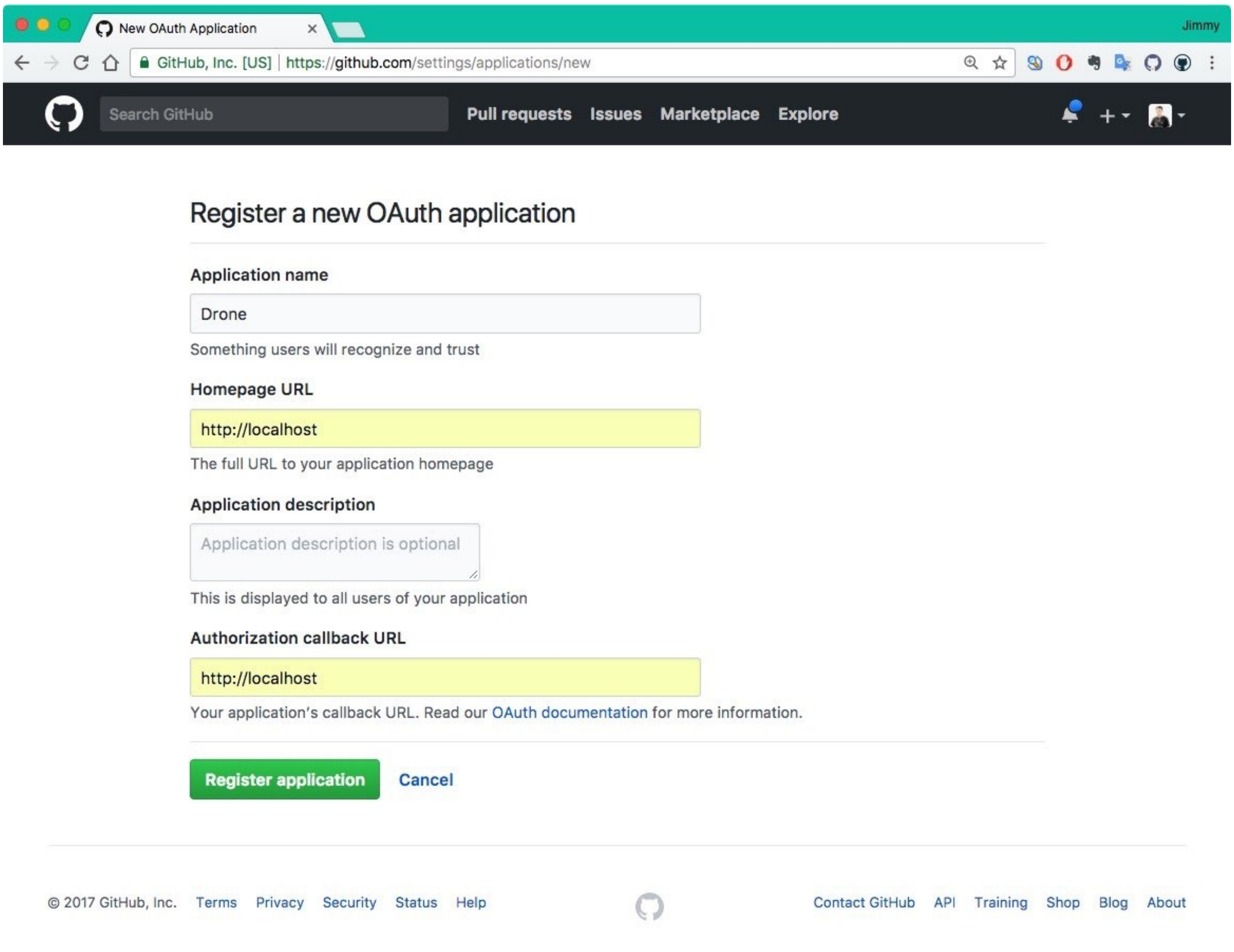
Drone是一个用Go语言开发的基于容器运行的持续集成软件。

配置GitHub

使用Drone对GitHub上的代码进行持续构建与发布，需要首先在GitHub上设置一个OAuth，如下：

1. 在Github上创建一个新的OAtuh应用

访问此頁面，创建新的OAuth应用。

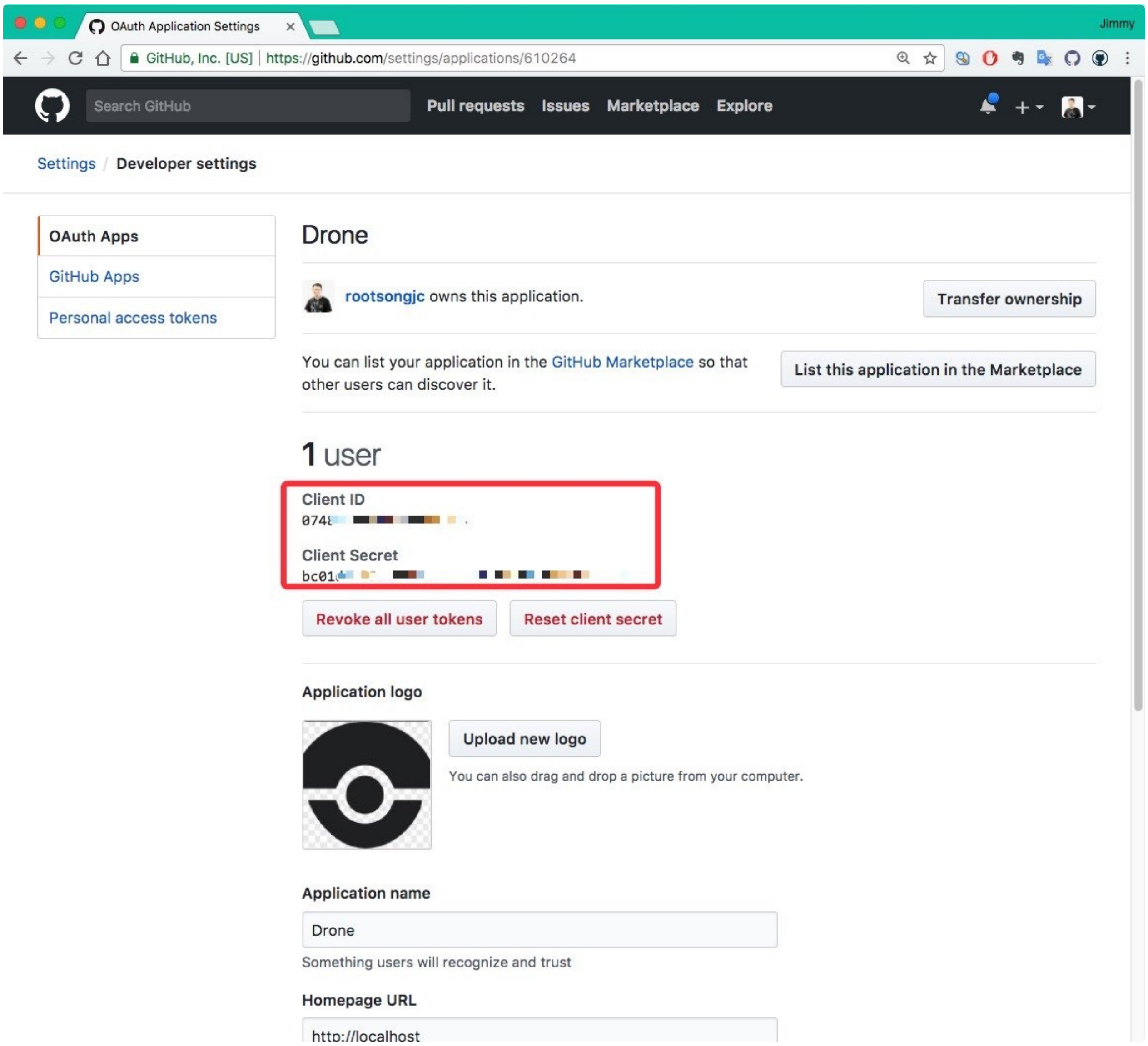


图片 - *OAuth*注册

填写应用程序的地址，因为是在本地与行，所以我们都填 `http://localhost` 。

2. 获取OAtuh Client ID和Client Secret

在注册完成后就可以获得如下图所示的OAuth Client ID和Client Secret，保存下来，我们后面要用到。



图片 - *OAuth key*

使用**docker-compose**单机运行

我们在本地环境，使用docker-compose，按照Drone官方安装文档安装配置Drone。

我们将代码托管在Github上，需要Drone可以持续集成和发布Github的代码，因此需要修改 `docker-compoe.yaml` 文件中的GitHub配置。

```yaml
version: '2'

services:
  drone-server:
    image: drone/drone:0.8

    ports:
      - 80:8000
      - 9000
    volumes:
      - /var/lib/drone:/var/lib/drone/
    restart: always
    environment:
      - DRONE_OPEN=true
      - DRONE_HOST=${DRONE_HOST}
      - DRONE_GITHUB=true
      - DRONE_GITHUB_CLIENT=${DRONE_GITHUB_CLIENT}
      - DRONE_GITHUB_SECRET=${DRONE_GITHUB_SECRET}
      - DRONE_SECRET=${DRONE_SECRET}

  drone-agent:
    image: drone/agent:0.8

    command: agent
    restart: always
    depends_on:
      - drone-server
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock
    environment:
      - DRONE_SERVER=drone-server:9000
      - DRONE_SECRET=${DRONE_SECRET}
```

- `/var/lib/drone` 是在本地挂载的目录，请确保该目录已存在，且可以被docker访问到，Mac下可以在docker的共享目录中配置。
- `DRONE_SECRET` 可以是一个随机的字符串，要确保 `drone-server` 与 `drone-client` 的 `DRONE_SECRET` 相同。
- `DRONE_GITHUB_CLIENT` 和 `DRONE_GITHUB_SECRET` 即在前面申请的OAuth的Client ID和Client Secret。

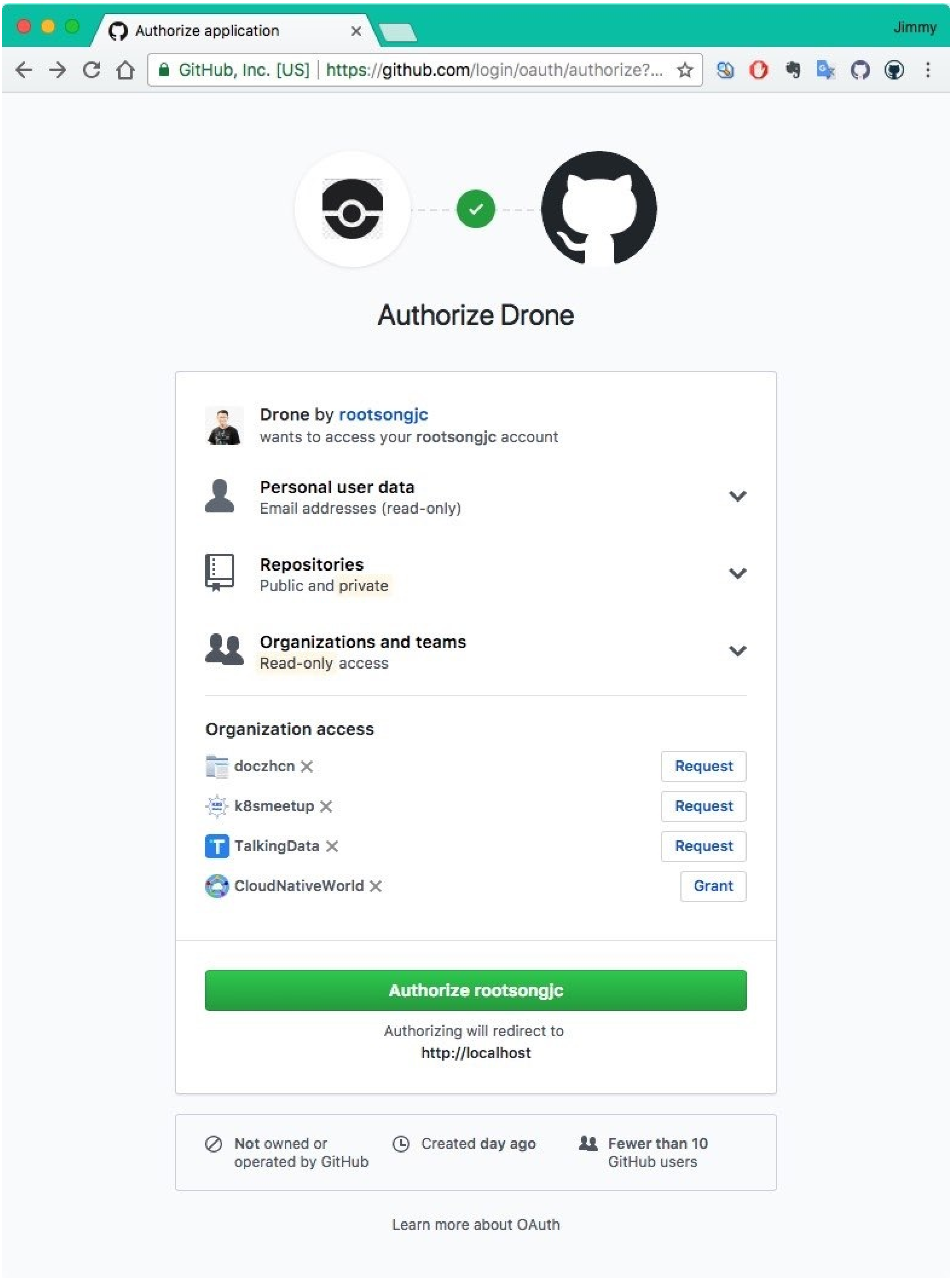
启动Drone

使用下面的命令在本地启动drone：

```
docker-compose up
```

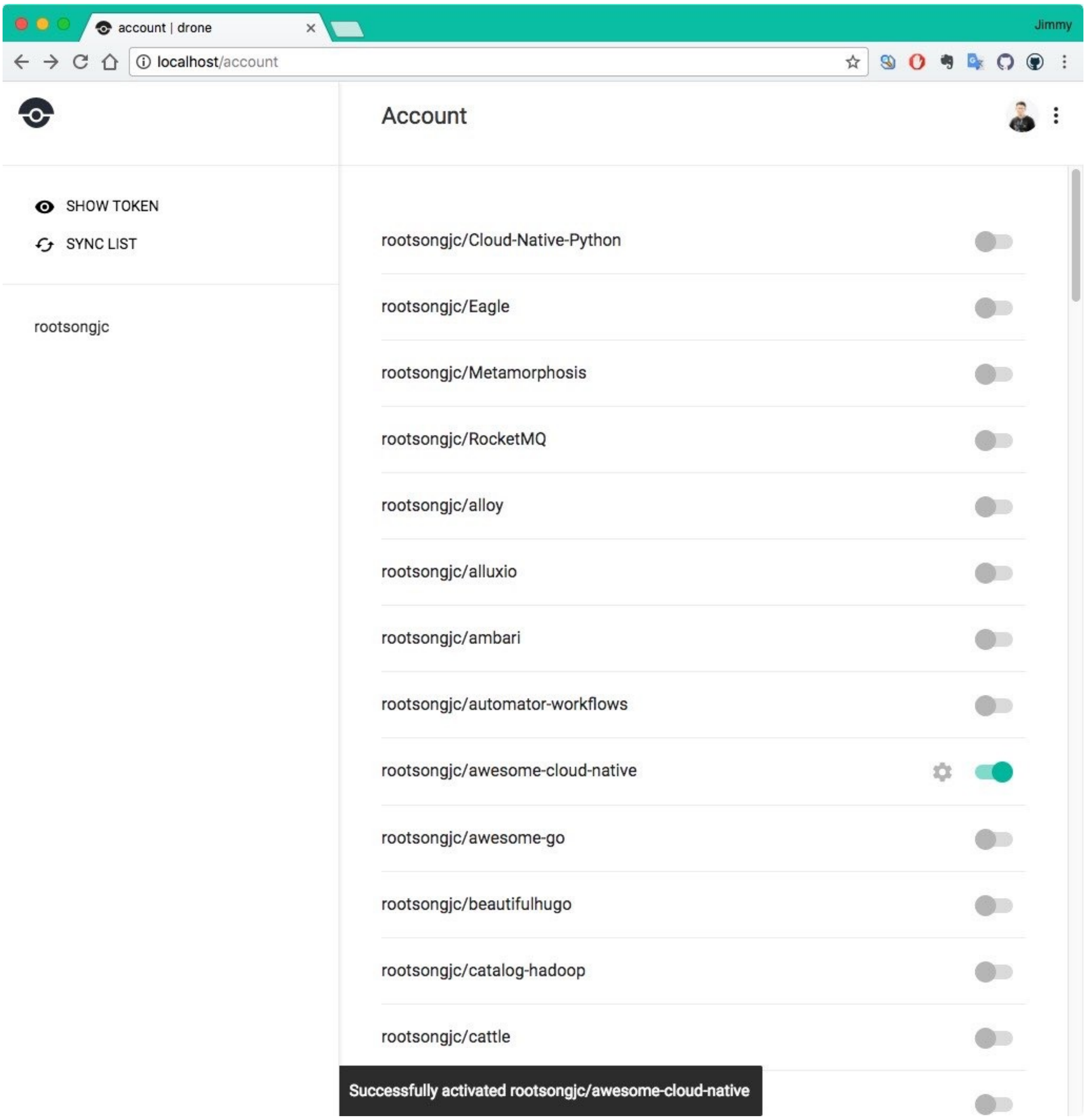
这样是在前台启动，加上 `-d` 参数就可以在后台启动。

访问 `http://localhost` 可以看到登陆画面。

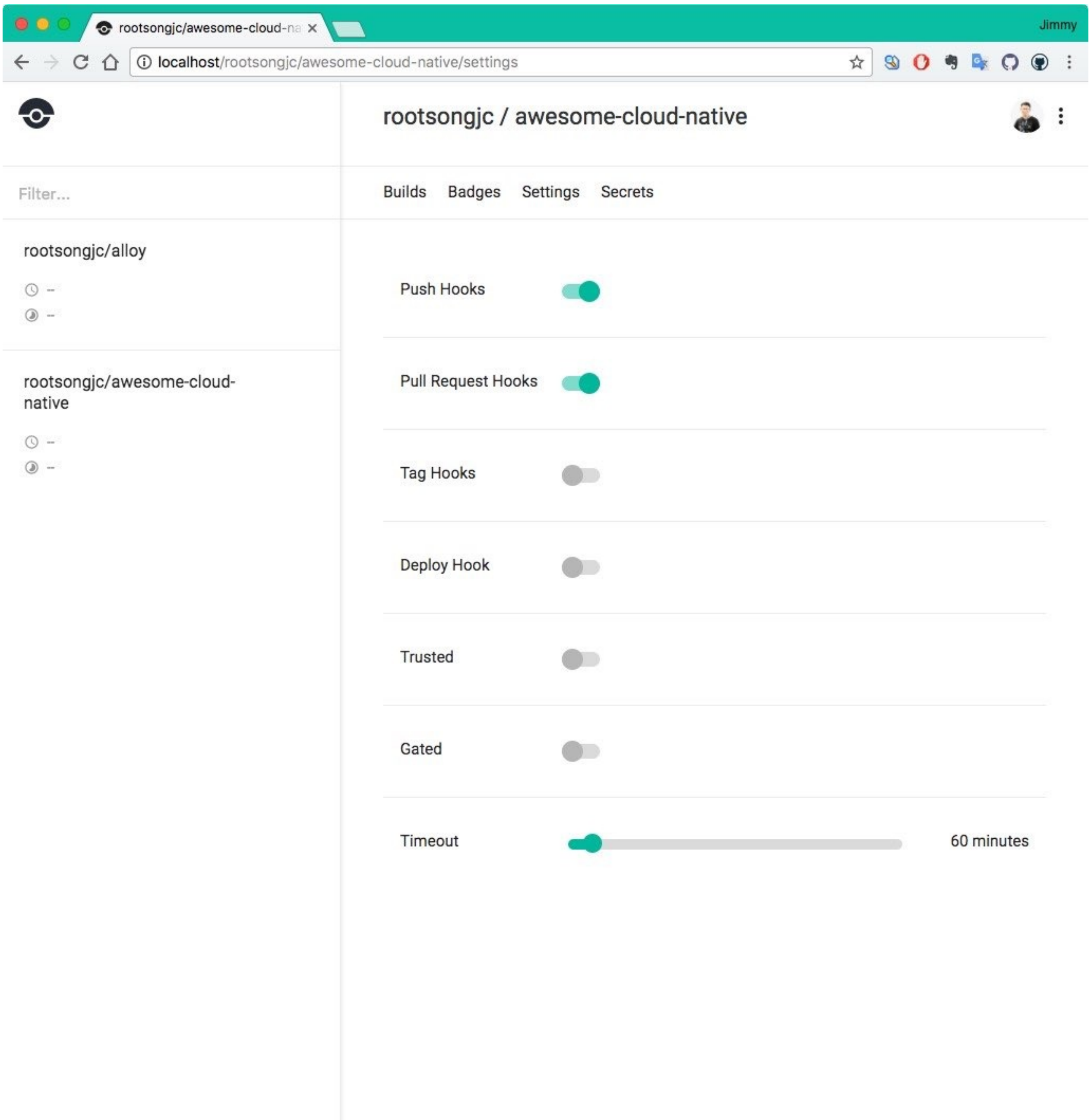


图片 - *Drone*登陆界面

授权后可以看到GitHub repo设置。



图片 - *Github*启用*repo*设置



图片 - *Github*单个*repo*设置

参考

- Drone Installation
- Github - Drone
- Drone 搭配 Kubernetes 升級應用程式版本 - blog.wu-boy.com

更新与升级

Kubernetes到目前为止基本保持三个月发行一个新版本的节奏，更新节奏可以说非常快，这一部分将主要跟踪kubernetes及其相关组件的更新与升级。

手动升级kubernetes集群

在我最开始写作本书的时候，kubernetes刚发布1.6.0版本，而kubernetes基本按照每三个月发布一个大版本的速度迭代，为了使用新特性和只支持新版本kubernetes的配套软件，升级kubernetes就迫在眉睫，在此我们使用替换kubernets的旧的二进制文件这种暴力的方式来升级测试集群，若升级生产集群还望三思。

另外，自kubernetes1.6版本之后发布的1.7和1.8版本又增加了一些新特性，参考：

- Kubernetes1.7更新日志
- Kubernetes1.8更新日志

目前kubernetes的官方文档上并没有详细的手动安装的集群如何升级的参考资料，只有两篇关于kubernetes集群升级的文档。

- 在ubuntu上如何使用juju升级：https://kubernetes.io/docs/getting-started-guides/ubuntu/upgrades/
- 使用kubeadm升级：https://kubernetes.io/docs/tasks/administer-cluster/kubeadm-upgrade-1-7/

手动升级的还没有详细的方案，大多是基于管理工具部署和升级，比如juju、kubeadm、kops、kubespray等。

manual upgrade/downgrade testing for Kubernetes 1.6 - google group，在这个Google group中讨论了kubernetes手动升级的问题，并给出了参考建议。

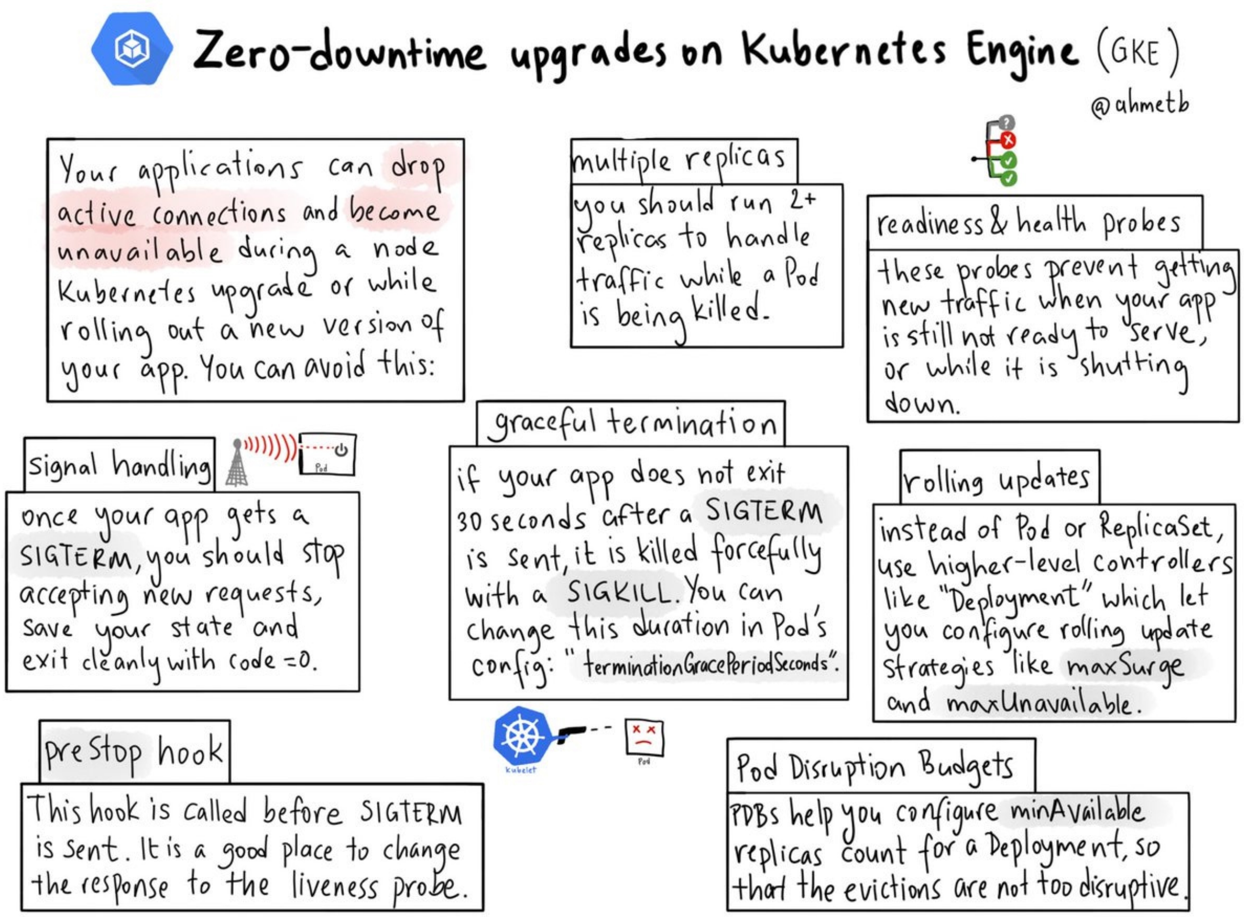
升级步骤

> 注意：该升级步骤是实验性的，建议在测试集群上使用，无法保证线上服务不中断，实际升级完成后无需对线上服务做任何操作。

大体上的升级步骤是，先升级master节点，然后再一次升级每台node节点。

升级建议

下图来自@ahmetb的Twitter，这是他对于0宕机时间的kubernetes集群升级建议。



图片 - *Kubernetes*零宕机时间升级建议

主要包括以下建议：

- 应用使用高级对象定义，如支持滚动更新的 `Deployment` 对象
- 应用要部署成多个实例
- 使用pod的preStop hook，加强pod的生命周期管理
- 使用就绪和健康检查探针来确保应用存活和及时阻拦应用流量的分发

准备

1. 备份kubernetes原先的二进制文件和配置文件。
2. 下载最新版本的kubernetes二进制包，如1.8.5版本，查看changelog，下载二进制包，我们使用的是kubernetes-server-linux-amd64.tar.gz，分发到集群的每个节点上。

升级master节点

停止master节点的进程

```
systemctl stop kube-apiserver
systemctl stop kube-scheduler
systemctl stop kube-controller-manager
systemctl stop kube-proxy
systemctl stop kubelet
```

使用新版本的kubernetes二进制文件替换原来老版本的文件，然后启动master节点上的进程：

```
systemctl start kube-apiserver
systemctl start kube-scheduler
systemctl start kube-controller-manager
```

因为我们的master节点同时也作为node节点，所有还要执行下面的"升级node节点"中的步骤。

升级node节点

关闭swap

```
# 临时关闭
swapoff -a

# 永久关闭，注释掉swap分区即可
vim /etc/fstab
#UUID=65c9f92d-4828-4d46-bf19-fb78a38d2fd1 swap
   swap    defaults        0 0
```

修改kubelet的配置文件

将kubelet的配置文件 `/etc/kubernetes/kublet` 配置文件中的 `KUBELET_API_SERVER="--api-servers=http://172.20.0.113:8080"` 行注释掉。

> 注意：：kubernetes1.7及以上版本已经没有该配置了，API server的地址写在了kubeconfig文件中。

停止node节点上的kubernetes进程：

```
systemctl stop kubelet
systemctl stop kube-proxy
```

使用新版本的kubernetes二进制文件替换原来老版本的文件，然后启动node节点上的进程：

```
systemctl start kubelet
systemctl start kube-proxy
```

启动新版本的kube-proxy报错找不到 `conntrack` 命令，使用 `yum install -y conntrack-tools` 命令安装后重启kube-proxy即可。

检查

到此升级完成，在master节点上检查节点状态：

```
NAME             STATUS    ROLES     AGE       VERSION
172.20.0.113     Ready     <none>    244d      v1.8.5
172.20.0.114     Ready     <none>    244d      v1.8.5
172.20.0.115     Ready     <none>    244d      v1.8.5
```

所有节点的状态都正常，再检查下原先的运行在kubernetes之上的服务是否正常，如果服务正常的话说明这次升级无误。

API版本变更适配

对于不同版本的Kubernetes，许多资源对象的API的版本可能会变更，下表列出了kubernetes1.5至1.9的API资源对象的版本演进：

Kubernetes资源对象的版本演进

Catalog	Kind	1.5		1.6		1.7		1.8		1.9	
		Group	Version	Group	Version	Group	Version	Group	Version	Group	Version
	Container	Core	v1	Core	v1	Core	v1	Core	v1	Core	v1
	CronJob	Batch	v2alpha1	Batch	v2alpha1	Batch	v2alpha1	Batch	v1beta1	Batch	v1beta1
	DaemonSet	Extensions	v1beta1	Extensions	v1beta1	Extensions	v1beta1	Apps	v1beta2	Apps	v1
	Deployment	Extensions	v1beta1	Apps	v1beta1	Apps	v1beta1	Apps	v1beta2	Apps	v1
Workloads	Job	Batch	v1	Batch	v1	Batch	v1	Batch	v1	Batch	v1
	Pod	Core	v1	Core	v1	Core	v1	Core	v1	Core	v1
	ReplicaSet	Extensions	v1beta1	Extensions	v1beta1	Extensions	v1beta1	Apps	v1beta2	Apps	v1
	ReplicationController	Core	v1	Core	v1	Core	v1	Core	v1	Core	v1
	StatefulSet	Apps	v1beta1	Apps	v1beta1	Apps	v1beta1	Apps	v1beta2	Apps	v1
	Endpoints	Core	v1	Core	v1	Core	v1	Core	v1	Core	v1
Discovery & Load Balancing	Ingress	Extensions	v1beta1	Extensions	v1beta1	Extensions	v1beta1	Extensions	v1beta1	Extensions	v1beta1
	Service	Core	v1	Core	v1	Core	v1	Core	v1	Core	v1
	ConfigMap	Core	v1	Core	v1	Core	v1	Core	v1	Core	v1
	Secret	Core	v1	Core	v1	Core	v1	Core	v1	Core	v1
Config & Storage	PersistentVolumeClaim	Core	v1	Core	v1	Core	v1	Core	v1	Core	v1
	StorageClass	Storage	v1beta1	Storage	v1beta1	Storage	v1	Storage	v1	Storage	v1
	Volume	Core	v1	Core	v1	Core	v1	Core	v1	Core	v1
	VolumeAttachment									Storage	v1alpha1
	ControllerRevision					Apps	v1beta1	Apps	v1beta2	Apps	v1
	CustomResourceDefinition							ApiExtensions	v1beta1	ApiExtensions	v1beta1
	Event	Core	v1	Core	v1	Core	v1	Core	v1	Core	v1
	LimitRange	Core	v1	Core	v1	Core	v1	Core	v1	Core	v1
	ExternalAdmissionHookConfiguration					AdmissionRegistration	v1alpha1	AdmissionRegistration	v1alpha1		
	HorizontalPodAutoscaler	Autoscaling	v1	Autoscaling	v1	Autoscaling	v1	Autoscaling	v1	Autoscaling	v1
	InitializerConfiguration					AdmissionRegistration	v1alpha1	AdmissionRegistration	v1alpha1	AdmissionRegistration	v1alpha1
Metadata	MutatingWebhookConfiguration									AdmissionRegistration	v1beta1
	ValidatingWebhookConfiguration									AdmissionRegistration	v1beta1
	PodTemplate	Core	v1	Core	v1	Core	v1	Core	v1	Core	v1
	PodDisruptionBudget	Policy	v1beta1	Policy	v1beta1	Policy	v1beta1	Policy	v1beta1	Policy	v1beta1
	ThirdPartyResource	Extensions	v1beta1	Policy	v1beta1	Extensions	v1beta1				
	PriorityClass							Scheduling	v1alpha1	Scheduling	v1alpha1
	PodPreset			Settings	v1alpha1	Settings	v1alpha1	Settings	v1alpha1	Settings	v1alpha1
	PodSecurityPolicy			Extensions	v1beta1	Extensions	v1beta1	Extensions	v1beta1	Extensions	v1beta1
	APIService					ApiRegistration	v1beta1	ApiRegistration	v1beta1	ApiRegistration	v1beta1
	Binding	Core	v1	Core	v1	Core	v1	Core	v1	Core	v1
	CertificateSigningRequest	Certificates	v1alpha1	Certificates	v1beta1	Certificates	v1beta1	Certificates	v1beta1	Certificates	v1beta1
	ClusterRole	RbacAuthorization	v1alpha1	RBAC	v1beta1	RBAC	v1beta1	RBAC	v1	RBAC	v1
	ClusterRoleBinding	RbacAuthorization	v1alpha1	RBAC	v1beta1	RBAC	v1beta1	RBAC	v1	RBAC	v1
	ComponentStatus	Core	v1	Core	v1	Core	v1	Core	v1	Core	v1
	LocalSubjectAccessReview	Authorization	v1beta1	Authorization	v1	Authorization	v1	Authorization	v1	Authorization	v1
	Namespace	Core	v1	Core	v1	Core	v1	Core	v1	Core	v1
	Node	Core	v1	Core	v1	Core	v1	Core	v1	Core	v1
Cluster	PersistentVolume	Core	v1	Core	v1	Core	v1	Core	v1	Core	v1
	ResourceQuota	Core	v1	Core	v1	Core	v1	Core	v1	Core	v1
	Role	RbacAuthorization	v1alpha1	RBAC	v1beta1	RBAC	v1beta1	RBAC	v1	RBAC	v1
	RoleBinding	RbacAuthorization	v1alpha1	RBAC	v1beta1	RBAC	v1beta1	RBAC	v1	RBAC	v1
	SelfSubjectAccessReview	Authorization	v1beta1	Authorization	v1	Authorization	v1	Authorization	v1	Authorization	v1
	SelfSubjectRulesReview							Authorization	v1	Authorization	v1
	ServiceAccount	Core	v1	Core	v1	Core	v1	Core	v1	Core	v1
	SubjectAccessReview	Authorization	v1beta1	Authorization	v1	Authorization	v1	Authorization	v1	Authorization	v1
	TokenReview	Authentication	v1beta1	Authorization	v1	Authorization	v1	Authorization	v1	Authorization	v1
	NetworkPolicy	Extensions	v1beta1	Extensions	v1beta1	Networking	v1	Networking	v1	Networking	v1

图片 - *Kuberentes API*对象的版本演进

当我们升级过后，可能出现资源对象的API变更后，原先的YAML文件无法使用的情况，因此需要对新版本的Kubernetes进行适配。对应的API版本转换工具：https://github.com/fleeto/kube-version-converter，可以将Kuberntes API对象转换到指定版本。

参考

- Cluster Upgrade #2524
- Upgrading self-hosted Kubernetes
- Upgrading Kubernetes - kops
- Upgrading kubeadm clusters from 1.6 to 1.7
- How to Upgrade a Kubernetes Cluster With No Downtime
- manual upgrade/downgrade testing for Kubernetes 1.6 - google group
- Notes/Instructions for Manual Upgrade Testing1.5 -> 1.6
- Upgrading Kubernetes in Kubespray

升级Dashboard

我们在kubernetes1.6的时候同时安装了dashboard插件，该插件也是基于kubernetes1.6版本开发的。如今kubernetes1.8版本业已发布，如何升级dashboard以获取新版中功能呢？

Dashboard的升级比较简单，因为它仅仅是一个前端应用，用来展现集群信息和与后端API交互，理论上只需要更新原先dashboard的yaml配置文件中的镜像就可以了，但是为了使用dashboard1.7版本中的用户登陆功能，还需要做一些额外的操作。

dashboard的更新日志请见release note，当前的最新版本为v1.7.1，下面将介绍将dashboard从v1.6.3升级到v1.7.1并开启用户登陆认证的详细步骤。

升级步骤

删除原来的版本

首先删除原来的dashboard资源：

```
kubectl delete -f dashboard/
```

将 `dashboard` 目录下的所有yaml文件中的资源全部删除，包括Deployment、service和角色绑定等。

部署新版本

我们使用官方的配置文件来安装，首先下载官方配置：

```
wget https://raw.githubusercontent.com/kubernetes/dashboard/master/src/deploy/recommended/kubernetes-dashboard.yaml
```

修改其中的两个镜像地址为我们的私有地址。

- gcr.io/google_containers/kubernetes-dashboard-init-amd64:v1.0.1
- gcr.io/google_containers/kubernetes-dashboard-amd64:v1.7.1

这个两个镜像可以同时从时速云上获取：

- index.tenxcloud.com/jimmy/kubernetes-dashboard-amd64:v1.7.1
- index.tenxcloud.com/jimmy/kubernetes-dashboard-init-amd64:v1.0.1

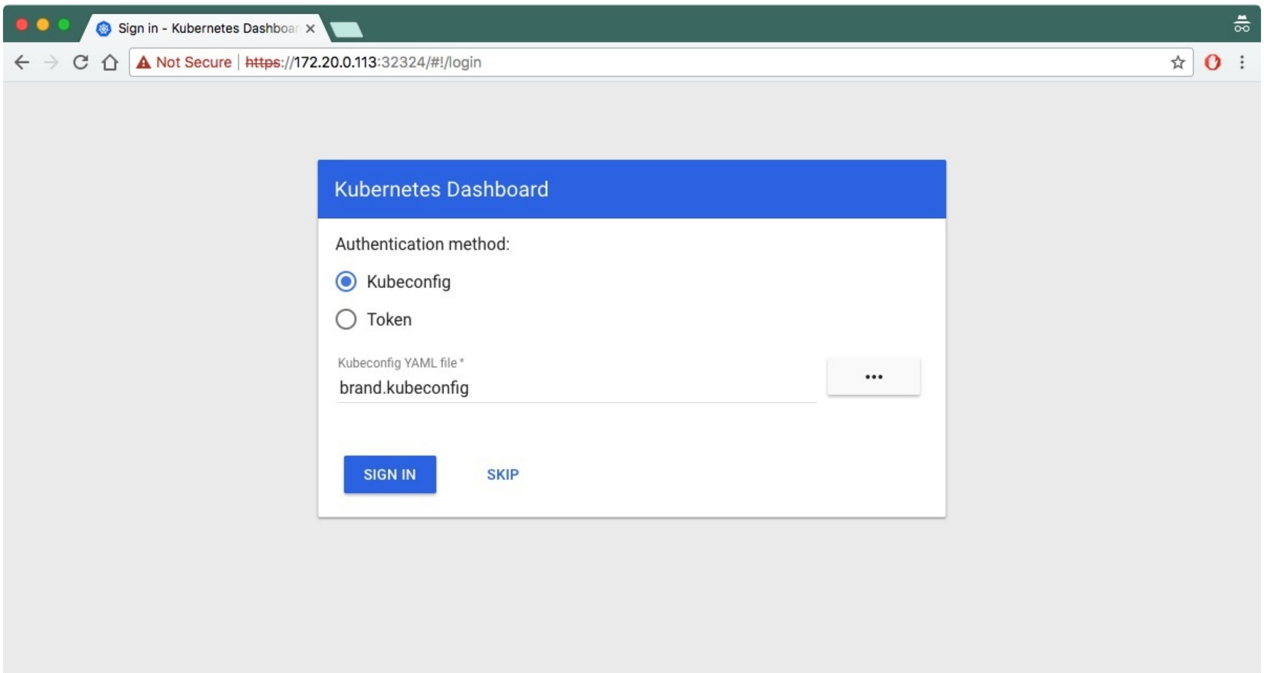
将service type设置为 `NodePort` ，修改后的yaml文件见kubernetes-dashboard.yaml，然后就可以部署新版本的dashboard了。

```
kubectl create -f kubernetes-dashboard.yaml
```

获取dashboard的外网访问端口：

```
kubectl -n kube-system get svc kubernetes-dashboard
NAME                    CLUSTER-IP       EXTERNAL-IP    PORT(S)
      AGE
kubernetes-dashboard   10.254.177.181   <nodes>        443:32324/
TCP    49m
```

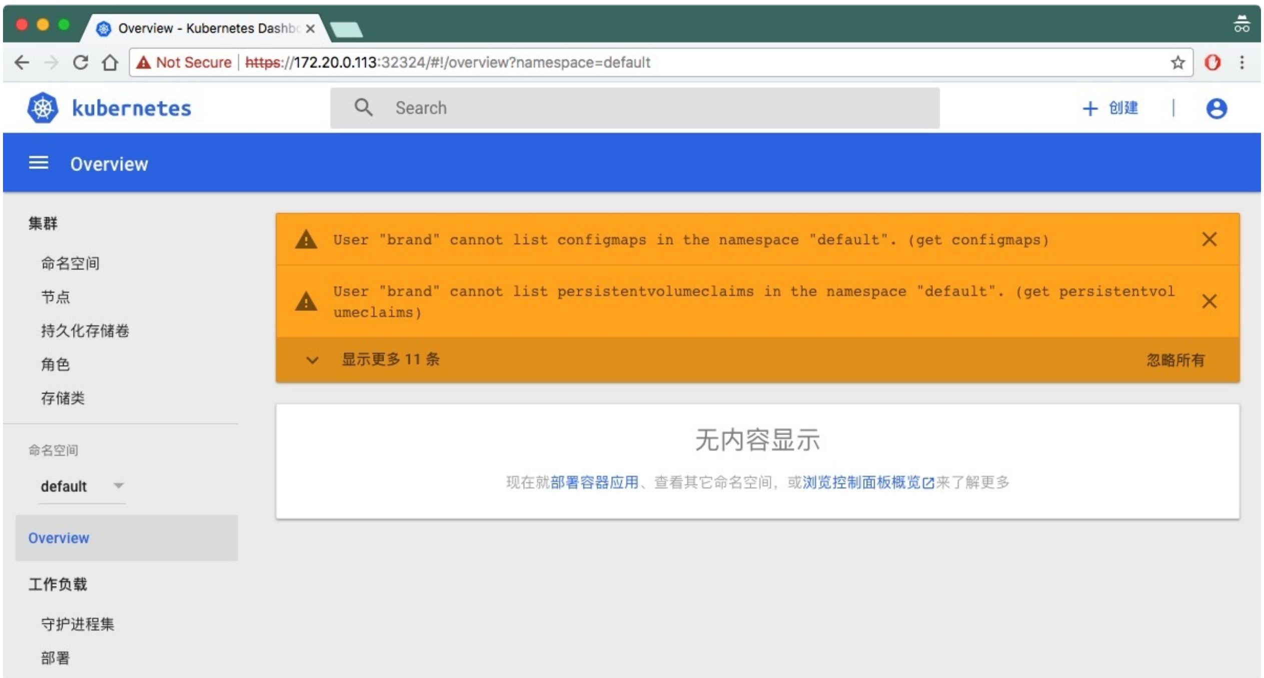
访问集群中的任何一个节点，即可打开dashboard登陆页面，如https://172.20.0.113:32324/（请使用https访问），支持使用 `kubeconfig` 和 `token` 两种的认证方式：



图片 - 登陆界面

选择本地的 `kubeconfig` 文件以登陆集群，`kubeconfig` 文件中包括登陆的用户名、证书和token信息。

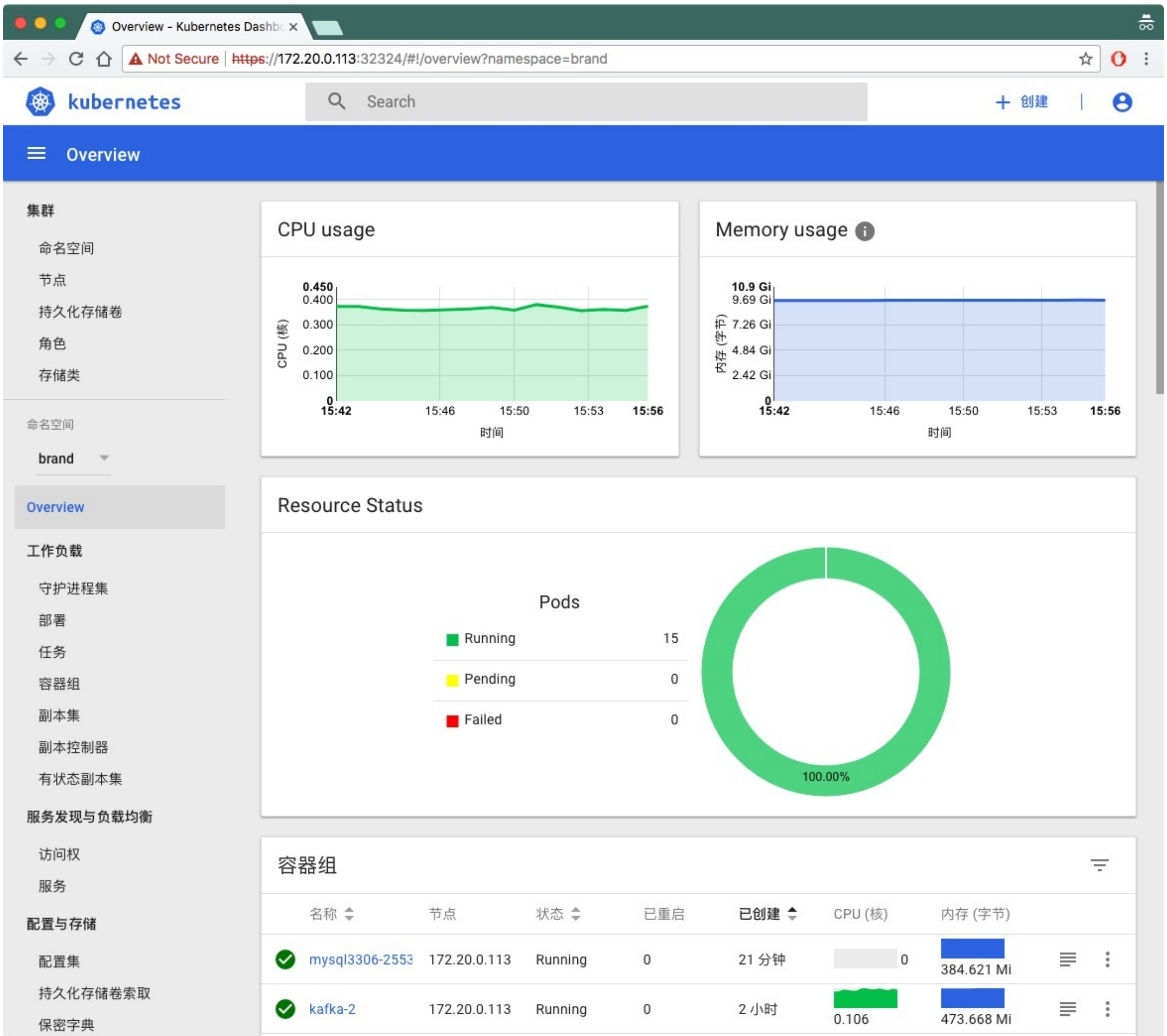
登陆之后首先看到的界面是这样的：



图片 - 首页

这是因为该用户没有对 `default` 命名空间的访问权限。

修改URL地址中的 `namespace` 字段为该用户有权限访问的命名空间如brand：https://172.20.0.113:32324/#!/overview?namespace=brand：



图片 - 用户空间

设置界面的语言

我们看到现在 dashboard 的页面都已经被汉化了，当前支持英文、中文简体、中文繁体、日语，根据浏览器的语言自动切换的。如果想要强制设置 dashboard 中显示的语言，需要在 dahsboard 的 Deployment yaml 配置中增加如下配置：

```
env:
  - name: ACCEPT_LANGUAGE
    value: english
```

关于 i18n 的设计文档请参考：https://github.com/kubernetes/dashboard/blob/master/docs/design/i18n.md

更简单的方式是，如果您使用的Chrome浏览器，则在浏览器中的配置中设置语言的顺序后刷新网页，dashboard将以您在Chrome中配置的首选语言显示。

身份认证

登陆 dashboard 的时候支持 kubeconfig 和 token 两种认证方式，kubeconfig 中也依赖 token 字段，所以生成 token 这一步是必不可少的。

下文分两块来讲解两种登陆认证方式：

- 为 brand 命名空间下的 brand 用户创建 kubeconfig 文件
- 为集群的管理员（拥有所有命名空间的 amdin 权限）创建 token

使用 kubeconfig

登陆dashboard的时候可以指定 `kubeconfig` 文件来认证用户权限，如何生成登陆dashboard时指定的 `kubeconfig` 文件请参考创建用户认证授权的kubeconfig文件。

> 注意我们生成的 kubeconfig 文件中没有 token 字段，需要手动添加该字段。

比如我们为 brand namespace 下的 brand 用户生成了名为 `brand.kubeconfig` 的 kubeconfig 文件，还要再该文件中追加一行 `token` 的配置（如何生成 token 将在下文介绍），如下所示：

```yaml
! brand.kubeconfig ×
1  apiVersion: v1
2  clusters:
3  - cluster:
4      certificate-authority-data: LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSURtakN
5      server: https://172.20.0.113:6443
6    name: kubernetes
7  contexts:
8  - context:
9      cluster: kubernetes
10     namespace: brand
11     user: brand
12   name: kubernetes
13 current-context: "kubernetes"
14 kind: Config
15 preferences: {}
16 users:
17 - name: brand
18   user:
19     client-certificate-data: LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUQwakNDQX
20     client-key-data: LS0tLS1CRUdJTiBSU0EgUFJJVkFURSBLRVktLS0tLQpNSUlFb3dJQkFBS0
21     token: a09bb459d67d876cf1829b4047394a5a
```

图片 - *kubeconfig*文件

这样就可以使用 `brand.kubeconfig` 文件来登陆dashboard了，而且只能访问和操作 `brand` 命名空间下的对象。

生成集群管理员的token

以下是为集群最高权限的管理员（可以任意操作所有namespace中的所有资源）生成token的步骤详解。

> 注意：登陆dashboard的时候token值是必须的，而kubeconfig文件是kubectl命令所必须的，kubectl命令使用的kubeconfig文件中可以不包含token信息。

需要创建一个admin用户并授予admin角色绑定，使用下面的yaml文件创建admin用户并赋予他管理员权限，然后可以通过token登陆dashbaord，该文件见admin-role.yaml。这种认证方式本质上是通过 Service Account 的身份认证加上 Bearer token 请求 API server 的方式实现，参考 Kubernetes 中的认证。

```yaml
kind: ClusterRoleBinding
apiVersion: rbac.authorization.k8s.io/v1beta1
metadata:
  name: admin
  annotations:
    rbac.authorization.kubernetes.io/autoupdate: "true"
roleRef:
  kind: ClusterRole
  name: cluster-admin
  apiGroup: rbac.authorization.k8s.io
subjects:
- kind: ServiceAccount
  name: admin
  namespace: kube-system
---
apiVersion: v1
kind: ServiceAccount
metadata:
  name: admin
  namespace: kube-system
  labels:
    kubernetes.io/cluster-service: "true"
    addonmanager.kubernetes.io/mode: Reconcile
```

然后执行下面的命令创建 serviceaccount 和角色绑定，对于其他命名空间的其他用户只要修改上述 yaml 中的 `name` 和 `namespace` 字段即可：

```
kubectl create -f admin-role.yaml
```

创建完成后获取secret和token的值。

运行下面的命令直接获取admin用户的token：

```
kubectl -n kube-system describe secret `kubectl -n kube-system get secret|grep admin-token|cut -d " " -f1`|grep "token:"|tr -s " "|cut -d " " -f2
```

手动获取

也可以执行下面的步骤来获取token值：

```
# 获取admin-token的secret名字
$ kubectl -n kube-system get secret|grep admin-token
admin-token-nwphb                                    kubernetes.io/service
-account-token   3         6m
# 获取token的值
$ kubectl -n kube-system describe secret admin-token-nwphb
Name:         admin-token-nwphb
Namespace:     kube-system
Labels:         <none>
Annotations:    kubernetes.io/service-account.name=admin
        kubernetes.io/service-account.uid=f37bd044-bfb3-11e7-87c
0-f4e9d49f8ed0

Type:    kubernetes.io/service-account-token

Data
====
namespace:    11 bytes
token:        非常长的字符串
ca.crt:        1310 bytes
```

在 dashboard 登录页面上有两种登录方式， `kubeconfig` 文件和 token （令牌），使用 token 登录可以直接使用上面输出中的那个非常长的字符串作为 token 登录，即可以拥有管理员权限操作整个kubernetes集群中的对象。对于 `kubeconfig` 文件登录方式，不能直接使用之前给 kubectl 生成的 `kubeconfig` 文件(`~/.kube/config`) 需要给它加一个 token 字段，您可以将这串 token 加到 admin 用户的 `kubeconfig` 文件中，继续使用 `kubeconfig` 登录，具体加的位置可以参考 `bootstrap-kubeconfig` 文件，两种认证方式任您选择。

注意：通过 `kubectl get secret xxx` 输出中的 token 值需要进行 `base64` 解码，在线解码工具 base64decode，Linux 和 Mac 有自带的 `base64` 命令也可以直接使用，输入 `base64` 是进行编码，Linux 中 `base64 -d` 表示解码，Mac 中使用 `base64 -D` ；通过 `kubectl describe secret xxx` 输出中的 token 不需要 `base64` 解码。

也可以使用 jsonpath 的方式直接获取 token 的值，如：

```
kubectl -n kube-system get secret admin-token-nwphb -o jsonpath={
.data.token}|base64 -d
```

注意我们使用了 base64 对其重新解码，因为 secret 都是经过 base64 编码的，如果直接使用 kubectl 中查看到的 `token` 值会认证失败，详见 secret 配置。关于 JSONPath 的使用请参考 JSONPath 手册。

注意：关于如何给其它namespace的管理员生成token请参考使用kubeconfig或token进行用户身份认证。

参考

- Dashboard log in mechanism #2093
- Accessing Dashboard 1.7.X and above
- Kubernetes dashboard UX for Role-Based Access Control
- How to sign in kubernetes dashboard? - StackOverflow
- JSONPath 手册
- Kubernetes 中的认证

领域应用

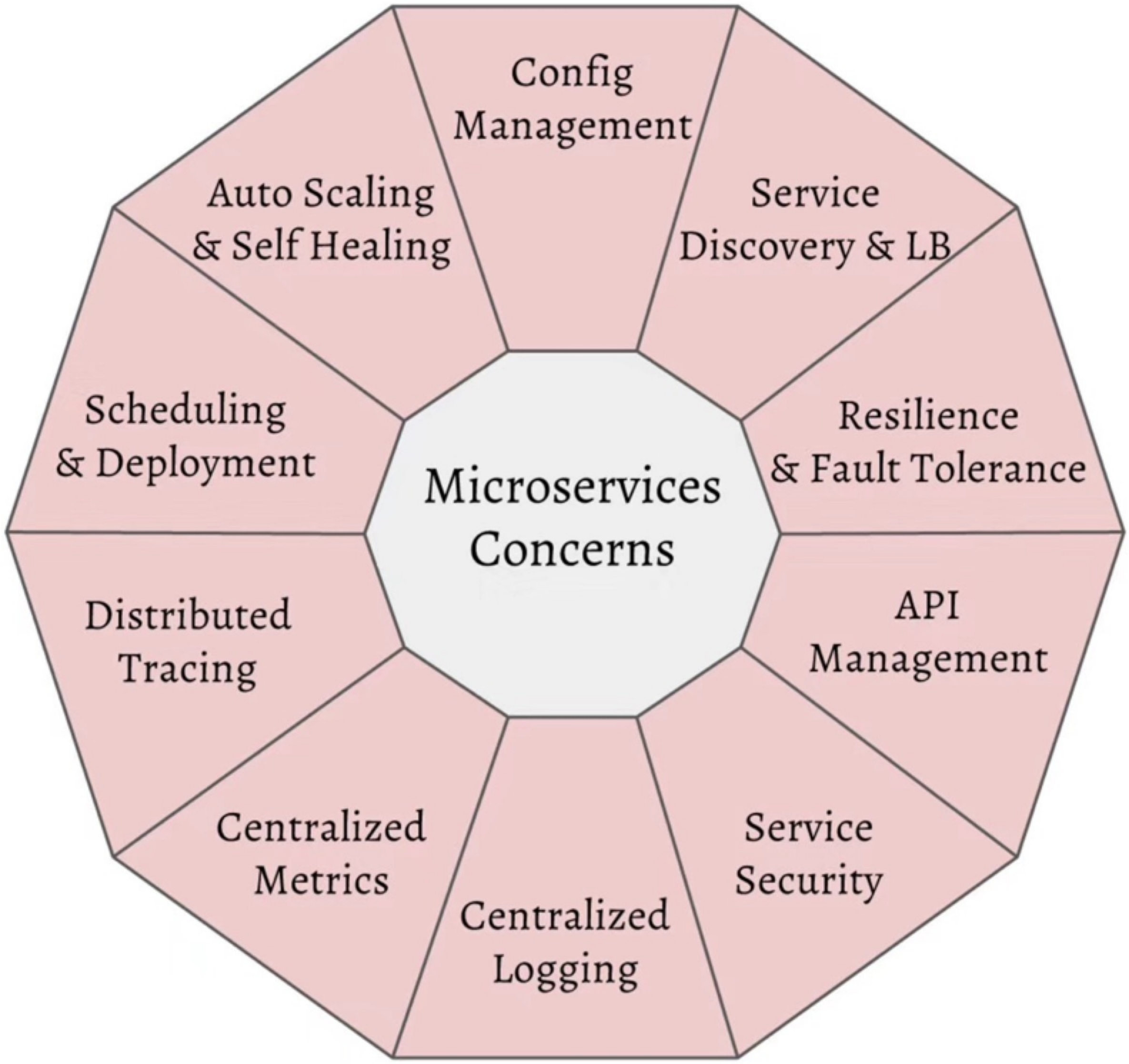
Kubernetes 和云原生应用在各个领域中的实践。

微服务架构

Kubernetes 设计之初就是按照 Cloud Native 的理念设计的，Cloud Native 中有个重要概念就是微服务的架构设计，当将单体应用拆分微服务后， 随着服务数量的增多，如何微服务进行管理以保证服务的 SLA 呢？为了从架构层面上解决这个问题，解放程序员的创造性，避免繁琐的服务发现、监控、分布式追踪等事务，Service mesh 应运而生。

微服务

下图是Bilgin Ibryam给出的微服务中应该关心的主题，图片来自RedHat Developers。



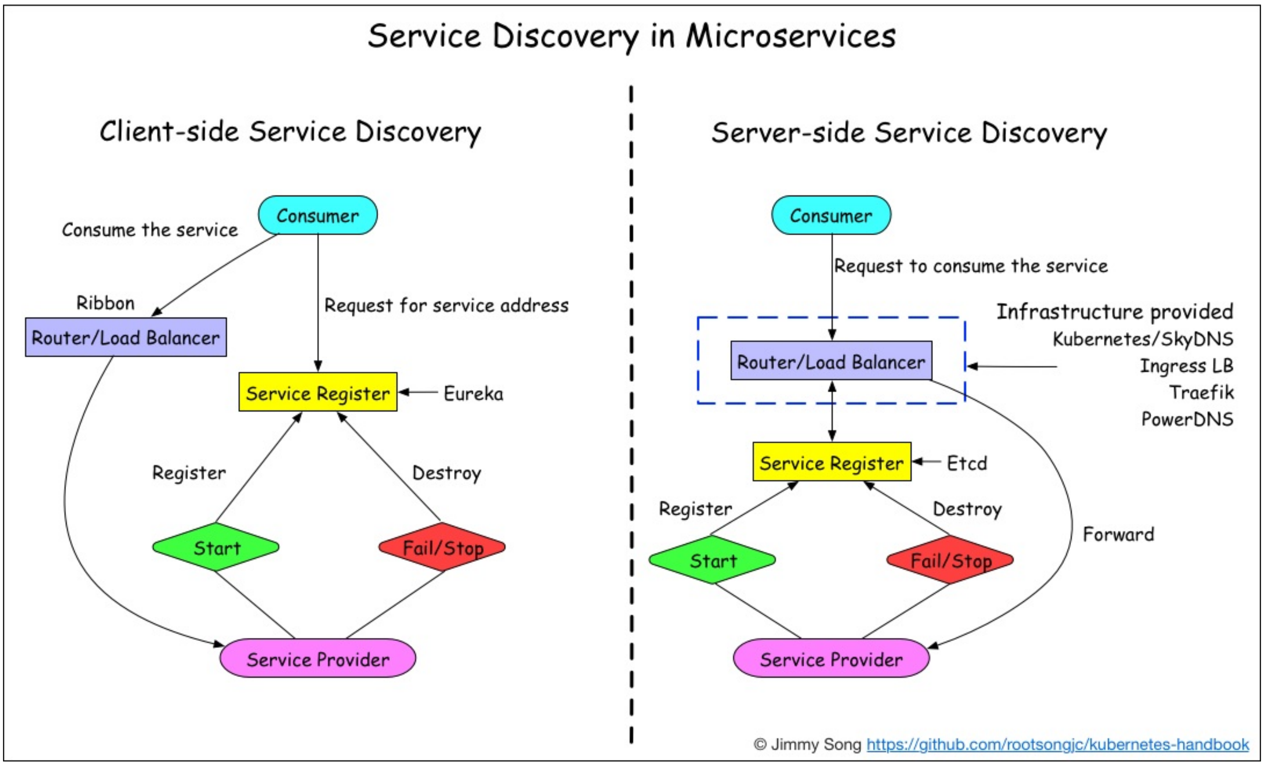
图片 - 微服务关注的部分

当前最成熟最完整的微服务框架可以说非Spring莫属，而Spring又仅限于Java语言开发，其架构本身又跟Kubernetes存在很多重合的部分，如何探索将Kubernetes作为微服务架构平台就成为一个热点话题。

微服务中的服务发现

在单体架构时，因为服务不会经常和动态迁移，所有服务地址可以直接在配置文件中配置，所以也不会有服务发现的问题。但是对于微服务来说，应用的拆分，服务之间的解耦，和服务动态扩展带来的服务迁移，服务发现就成了微服务中的一个关键问题。

服务发现分为客户端服务发现和服务端服务发现两种，架构如下图所示。



图片 - 微服务中的服务发现

这两种架构都各有利弊，我们拿客户端服务发现软件Eureka和服务端服务发现架构Kubernetes/SkyDNS+Ingress LB+Traefik+PowerDNS为例说明。

服务发现方案	Pros	Cons
Eureka	使用简单，适用于java语言开发的项目，比服务端服务发现少一次网络跳转	对非Java语言的支持不够好，Consumer需要内置特定的服务发现客户端和发现逻辑
Kubernetes	Consumer无需关注服务发现具体细节，只需知道服务的DNS域名即可，支持异构语言开发	需要基础设施支撑，多了一次网络跳转，可能有性能损失

Eureka 也不是单独使用的，一般会配合 ribbon 一起使用，ribbon 作为路由和负载均衡。

Ribbon提供一组丰富的功能集：

- 多种内建的负载均衡规则：
    - Round-robin 轮询负载均衡
    - 平均加权响应时间负载均衡
    - 随机负载均衡
    - 可用性过滤负载均衡（避免跳闸线路和高并发链接数）
    - 自定义负载均衡插件系统
- 与服务发现解决方案的可拔插集成（包括Eureka）
- 云原生智能，例如可用区亲和性和不健康区规避
- 内建的故障恢复能力

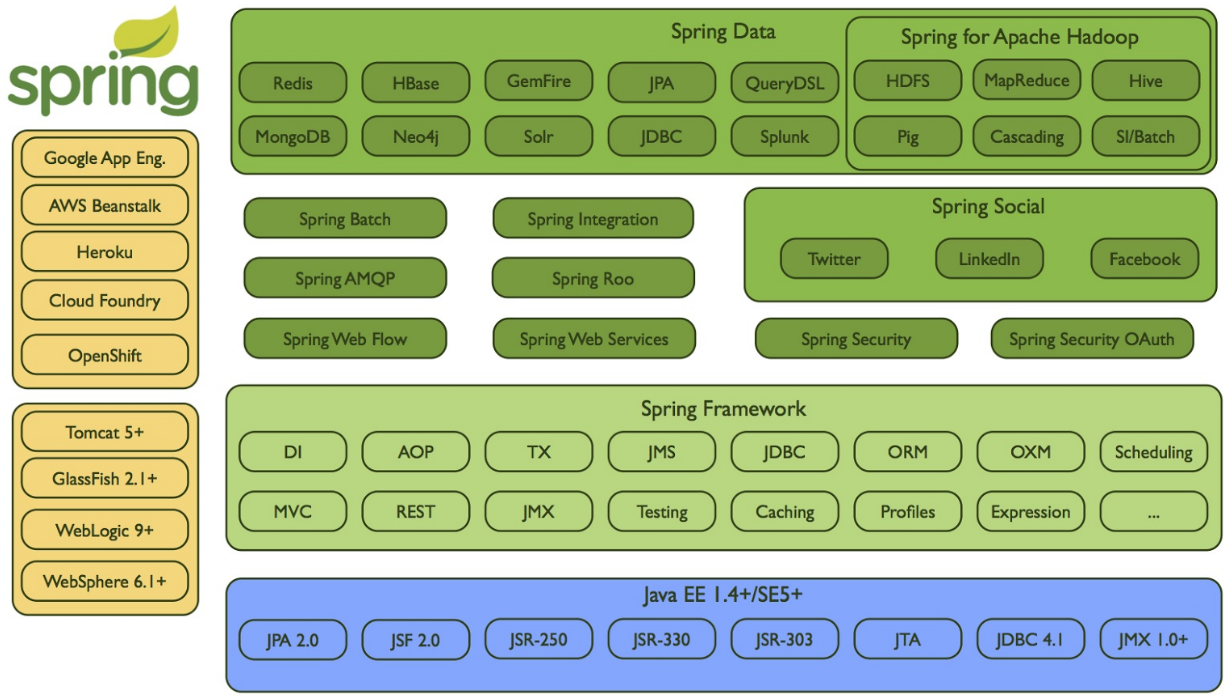
参考

- 谈服务发现的背景、架构以及落地方案

使用Java构建微服务并发布到Kubernetes平台

Java作为多年的编程语言届的No.1（使用人数最多，最流行），使用它来构建微服务的人也不计其数，Java的微服务框架Spring中的Spring Boot和Spring Cloud已成为当前最流行的微服务框架。

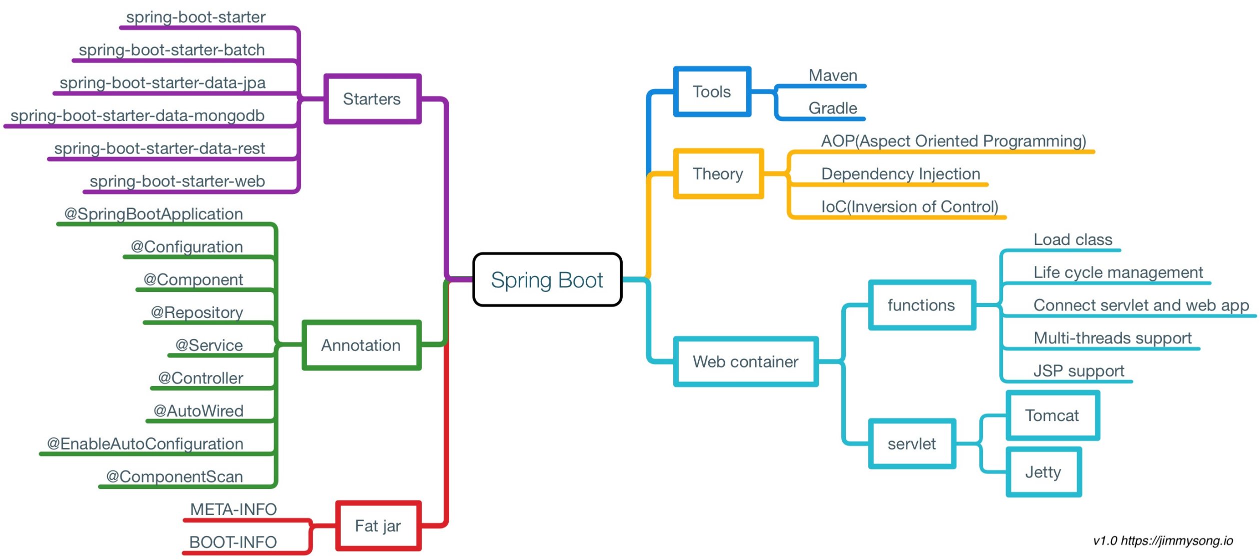
下面是Sping技术栈所包含的技术框架图。



图片 - *Spring*技术栈

当然如果在Kubernetes中运行Java语言构建的微服务应用，我们不会使用上图中所有的技术，本节将主要讲解如何使用Spring Boot构建微服务应用。

下图是Spring Boot的一些知识点。



图片 - *Spring Boot*的知识点

Spring Boot是Spring框架的一部分，关于Spring的核心技术请参考Spring core technologies - spring.io。

Updated at 2018-05-09 14:15:47

Spring Boot快速开始指南

Spring Boot已成为当今最流行的微服务开发框架，本文是如何使用Spring Boot快速开始Web微服务开发的指南，我们将使创建一个可运行的包含内嵌Web容器（默认使用的是Tomcat）的可运行Jar包。

Spring Boot旨在简化创建产品级的Spring应用和服务，简化了配置文件，使用嵌入式web服务器，含有诸多开箱即用微服务功能，可以和spring cloud联合部署。

传统的Spring应用程序需要配置大量的XML文件才能运行，而使用Spring Boot只需极少的配置，就可以快速获得一个正常运行的Spring应用程序，而这些配置使用的都是注解的形式，不需要再配置XML。

与Go语言的应用不同，我们知道所有的Java Web应用都必须放在servlet容器（不是像docker容器的那种容器），如Tomcat、Jetty等。Servlet容器被定位为托管web应用程序的高可用组件。关于Servlet的教程请参考Servlet教程 | runoob.com。

Spring的基本原理

Spring是一套Java开发框架，框架的作用就是为了减少代码的冗余和模块之间的偶尔，使代码逻辑更加清晰，主要是用了AOP（Aspect Oriented Programming，面向切面编程）和IoC（Inversion of Control，控制反转）容器的思想，其中AOP是利用了Java的反射机制实现的。为了便于理解AOP可以参考一个简单的Spring的AOP例子。

准备环境

在开始Spring Boot开发之前，需要先确认您的电脑上已经有以下环境：

- JDK8
- Maven3.0+
- Intellij IDEA

JDK最好使用JDK8版本，Maven和IDEA的安装都十分简单，Maven的仓库配置有必要说一下。

配置Maven

在安装好Maven之后，默认的 `~/.m2` 目录下是没有maven仓库配置文件 `settings.xml` 的，默认使用的是官方的仓库，访问速度会非常慢，我们需要配置下国内的仓库。

创建 `~/.m2/settings.xml` 文件，文件内容如下：

```
<?xml version="1.0"?>
<settings>
  <mirrors>
        <mirror>
            <id>alimaven</id>
            <name>aliyun maven</name>
            <url>http://maven.aliyun.com/nexus/content/groups/pu
blic/</url>
            <mirrorOf>central</mirrorOf>
        </mirror>
  </mirrors>
  <profiles>
    <profile>
       <id>nexus</id>
        <repositories>
            <repository>
                <id>nexus</id>
                <name>local private nexus</name>
                <url>http://maven.oschina.net/content/groups/pub
lic/</url>
                <releases>
                    <enabled>true</enabled>
                </releases>
                <snapshots>
                    <enabled>false</enabled>
                </snapshots>
            </repository>
        </repositories>

        <pluginRepositories>
            <pluginRepository>
            <id>nexus</id>
            <name>local private nexus</name>
            <url>http://maven.oschina.net/content/groups/public/
</url>
            <releases>
                <enabled>true</enabled>
            </releases>
            <snapshots>
                <enabled>false</enabled>
            </snapshots>
            </pluginRepository>
        </pluginRepositories>
    </profile></profiles>
</settings>
```

其中使用的是阿里云的mirror，国内的下载速度非常快。

创建第一个Spring Boot应用

我们可以使用以下两种方式创建Spring Boot应用：

- springboot
- maven

使用springboot命令创建Spring Boot应用

首先需要安装 `springboot` 命令行工具。

```
brew tap pivotal/tap
brew install springboot
```

使用下面的命令创建应用。

```
spring init --build maven --groupId com.example --version 0.0.1-
SNAPSHOT --java-version 1.8 --dependencies web --name myproject
myproject
```

- `--build` 使用maven编译或者是gradle
- `--groupId` 和 `--version` 与maven的 `pom.xml` 中的设置对应
- `--dependencies` 可以指定多个，如 `web` 、 `jpa` 、 `security` 等starter

执行上述命令后，将创建如下的目录结构：

```
.
└── myproject
    ├── mvnw
    ├── mvnw.cmd
    ├── pom.xml
    └── src
        ├── main
        │   ├── java
        │   │   └── com
        │   │       └── example
        │   │           └── myproject
        │   │               └── MyprojectApplication.java
        │   └── resources
        │       ├── application.properties
        │       ├── static
        │       └── templates
        └── test
            └── java
                └── com
                    └── example
                        └── myproject
                            └── MyprojectApplicationTests.java

15 directories, 6 files
```

运行默认的示例应用。

```
mvn spring-boot:run
```

第一次运行需要下载依赖包所以会比较耗费时间，以后每次编译运行速度就会很快。

在浏览器中访问将看到如下输出：

```
Whitelabel Error Page
This application has no explicit mapping for /error, so you are
seeing this as a fallback.

Mon Mar 12 16:26:42 CST 2018
There was an unexpected error (type=Not Found, status=404).
No message available
```

使用Maven创建Spring Boot应用

使用Maven创建Spring Boot应用需要执行以下步骤：

1. 创建Maven工程所需的 `pom.xml` 文件
2. 生成Maven工程
3. 编译打包发布

创建pom.xml

为Maven项目构建创建 `pom.xml` 文件，内容如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"       xsi="ht
tp://www.w3.org/2001/XMLSchema-instance"
        schemaLocation="http://maven.apache.org/POM/4.0.0 http:/
/maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.example</groupId>
    <artifactId>myproject</artifactId>
    <version>0.0.1-SNAPSHOT</version>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>1.4.1.BUILD-SNAPSHOT</version>
    </parent>

    <repositories>
        <repository>
            <id>spring-snapshots</id>
            <url>http://repo.spring.io/snapshot</url>
            <snapshots><enabled>true</enabled></snapshots>
        </repository>
        <repository>
            <id>spring-milestones</id>
            <url>http://repo.spring.io/milestone</url>
        </repository>
    </repositories>
    <pluginRepositories>
        <pluginRepository>
            <id>spring-snapshots</id>
            <url>http://repo.spring.io/snapshot</url>
        </pluginRepository>
        <pluginRepository>
            <id>spring-milestones</id>
            <url>http://repo.spring.io/milestone</url>
        </pluginRepository>
    </pluginRepositories>
    <!-- 添加classpath依赖 -->
    <dependencies>
        <dependency>
                <groupId>org.springframework.boot</groupId>
```

```
                <artifactId>spring-boot-starter-web</artifactId>
            </dependency>
            <!-- 开发者工具，当classpath下有文件更新自动触发应用重启 -->
            <dependency>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-devtools</artifactId>
                <optional>true</optional>
            </dependency>
        </dependencies>
        <!-- maven编译插件，用于创建可执行jar包 -->
        <build>
            <plugins>
                <plugin>
                    <groupId>org.springframework.boot</groupId>
                    <artifactId>spring-boot-maven-plugin</artifactId>

                </plugin>
            </plugins>
        </build>
    </project>
```

现在执行 `mvn dependency:tree` 可以看到项目中的依赖关系。

```
com.example:myproject:jar:0.0.1-SNAPSHOT
\- org.springframework.boot:spring-boot-starter-web:jar:1.4.1.BU
ILD-SNAPSHOT:compile
   +- org.springframework.boot:spring-boot-starter:jar:1.4.1.BUI
LD-SNAPSHOT:compile
   |  +- org.springframework.boot:spring-boot:jar:1.4.1.BUILD-SN
APSHOT:compile
   |  +- org.springframework.boot:spring-boot-autoconfigure:jar:
1.4.1.BUILD-SNAPSHOT:compile
   |  +- org.springframework.boot:spring-boot-starter-logging:ja
r:1.4.1.BUILD-SNAPSHOT:compile
   |  |  +- ch.qos.logback:logback-classic:jar:1.1.7:compile
   |  |  |  +- ch.qos.logback:logback-core:jar:1.1.7:compile
   |  |  |  \- org.slf4j:slf4j-api:jar:1.7.21:compile
   |  |  +- org.slf4j:jcl-over-slf4j:jar:1.7.21:compile
   |  |  +- org.slf4j:jul-to-slf4j:jar:1.7.21:compile
   |  |  \- org.slf4j:log4j-over-slf4j:jar:1.7.21:compile
   |  +- org.springframework:spring-core:jar:4.3.3.RELEASE:compi
le
   |  \- org.yaml:snakeyaml:jar:1.17:runtime
   +- org.springframework.boot:spring-boot-starter-tomcat:jar:1.
4.1.BUILD-SNAPSHOT:compile
   |  +- org.apache.tomcat.embed:tomcat-embed-core:jar:8.5.5:com
pile
   |  +- org.apache.tomcat.embed:tomcat-embed-el:jar:8.5.5:compi
le
   |  \- org.apache.tomcat.embed:tomcat-embed-websocket:jar:8.5.
5:compile
   +- org.hibernate:hibernate-validator:jar:5.2.4.Final:compile
   |  +- javax.validation:validation-api:jar:1.1.0.Final:compile
   |  +- org.jboss.logging:jboss-logging:jar:3.3.0.Final:compile
   |  \- com.fasterxml:classmate:jar:1.3.1:compile
   +- com.fasterxml.jackson.core:jackson-databind:jar:2.8.3:comp
ile
   |  +- com.fasterxml.jackson.core:jackson-annotations:jar:2.8.
3:compile
   |  \- com.fasterxml.jackson.core:jackson-core:jar:2.8.3:compi
le
   +- org.springframework:spring-web:jar:4.3.3.RELEASE:compile
   |  +- org.springframework:spring-aop:jar:4.3.3.RELEASE:compil
e
   |  +- org.springframework:spring-beans:jar:4.3.3.RELEASE:comp
ile
   |  \- org.springframework:spring-context:jar:4.3.3.RELEASE:co
mpile
   \- org.springframework:spring-webmvc:jar:4.3.3.RELEASE:compil
e
      \- org.springframework:spring-expression:jar:4.3.3.RELEASE
:compile
```

这其中包括Tomcat web服务器和Spring Boot自身。

Spring Boot 推荐的基础 POM 文件

名称	说明
spring-boot-starter	核心 POM，包含自动配置支持、日志库和对 YAML 配置文件的支持。
spring-boot-starter-amqp	通过 spring-rabbit 支持 AMQP。
spring-boot-starter-aop	包含 spring-aop 和 AspectJ 来支持面向切面编程（AOP）。
spring-boot-starter-batch	支持 Spring Batch，包含 HSQLDB。
spring-boot-starter-data-jpa	包含 spring-data-jpa、spring-orm 和 Hibernate 来支持 JPA。
spring-boot-starter-data-mongodb	包含 spring-data-mongodb 来支持 MongoDB。
spring-boot-starter-data-rest	通过 spring-data-rest-webmvc 支持以 REST 方式暴露 Spring Data 仓库。
spring-boot-starter-jdbc	支持使用 JDBC 访问数据库。
spring-boot-starter-security	包含 spring-security。
spring-boot-starter-test	包含常用的测试所需的依赖，如 JUnit、Hamcrest、Mockito 和 spring-test 等。
spring-boot-starter-velocity	支持使用 Velocity 作为模板引擎。
spring-boot-starter-web	支持 Web 应用开发，包含 Tomcat 和 spring-mvc。
spring-boot-starter-websocket	支持使用 Tomcat 开发 WebSocket 应用。
spring-boot-starter-ws	支持 Spring Web Services。
spring-boot-starter-actuator	添加适用于生产环境的功能，如性能指标和监测等功能。
spring-boot-starter-remote-shell	添加远程 SSH 支持。
spring-boot-starter-jetty	使用 Jetty 而不是默认的 Tomcat 作为应用服务器。

spring-boot-starter-log4j	添加 Log4j 的支持。
spring-boot-starter-logging	使用 Spring Boot 默认的日志框架 Logback。
spring-boot-starter-tomcat	使用 Spring Boot 默认的 Tomcat 作为应用服务器。

所有这些 POM 依赖的好处在于为开发 Spring 应用提供了一个良好的基础。Spring Boot 所选择的第三方库是经过考虑的，是比较适合产品开发的选择。但是 Spring Boot 也提供了不同的选项，比如日志框架可以用 Logback 或 Log4j，应用服务器可以用 Tomcat 或 Jetty。

生成Maven工程

对于普通的Java项目或者Java Web项目可以使用下面的命令创建maven结构：

```
mvn archetype:generate -DgroupId=com.example -DartifactId=myproj
ect -DarchetypeArtifactId=maven-archetype-webapp -DinteractiveMo
de=false
```

下表是以上参数的使用说明：

参数	说明
mvn archetype:generate	固定格式
-DgroupId	组织标识（包名）
-DartifactId	项目名称
-DarchetypeArtifactId	指定ArchetypeId，maven-archetype-quickstart，创建一个Java Project；maven-archetype-webapp，创建一个Web Project
-DinteractiveMode	是否使用交互模式

这将生成以下的目录结构：

```
.
└── myproject
    ├── pom.xml
    └── src
        └── main
            ├── resources
            └── webapp
                ├── WEB-INF
                │   └── web.xml
                └── index.jsp

6 directories, 3 files
```

对于Spring Boot项目，无法使用 `mvn` 命令直接生成，需要手动创建目录：

```
mkdir -p src/main/java
```

创建示例代码

创建 `src/main/java/Example.java` 文件内容如下：

```
import org.springframework.boot.*;
import org.springframework.boot.autoconfigure.*;
import org.springframework.stereotype.*;
import org.springframework.web.bind.annotation.*;

@RestController
@EnableAutoConfiguration
public class Example {

    @RequestMapping("/")
    String home() {
        return "Hello World!";
    }

    public static void main(String[] args) throws Exception {
        SpringApplication.run(Example.class, args);
    }

}
```

- `@RestController` 注解告诉Spring以字符串的形式渲染结果，并直接返回给调用者。
- `@EnableAutoConfiguration` 注解告诉Spring Boot根据添加的jar依赖猜测你想如何配置Spring。由于 `spring-boot-starter-web` 添加了Tomcat和Spring

MVC，所以auto-configuration将假定你正在开发一个web应用，并对Spring进行相应地设置。
- `@RequestMapping` 注解提供路由信息，它告诉Spring任何来自"/"路径的HTTP请求都应该被映射到 `home` 方法。

注：`@RestController` 和 `@RequestMapping` 是Spring MVC中的注解（它们不是Spring Boot的特定部分）。

编译和发布

运行该项目有以下两种方式。

方式**1**：直接**mvn**命令运行

```
mvn spring-boot:run
```

方式**2**：编译打包成可执行**jar**包

```
mvn package
java -jar target/myproject-0.0.1-SNAPSHOT.jar
```

不论使用哪种方式编译，访问可以看到web页面上显示 `Hello world!` 。

在 `target` 目录下，你应该还能看到一个很小的名为 `myproject-0.0.1-SNAPSHOT.jar.original` 的文件，这是在Spring Boot重新打包前，Maven创建的原始jar文件。实际上可运行jar包中包含了这个小的jar包。

参考

- Spring官方网站
- Spring core technologies - spring.io
- Spring Boot——开发新一代Spring Java应用
- Spring MVC快速入门教程
- Spring Boot Reference Guide中文翻译 -《Spring Boot参考指南》
- 使用 Spring Boot 快速构建 Spring 框架应用
- maven3常用命令、java项目搭建、web项目搭建详细图解
- Servlet教程 - runoob.com

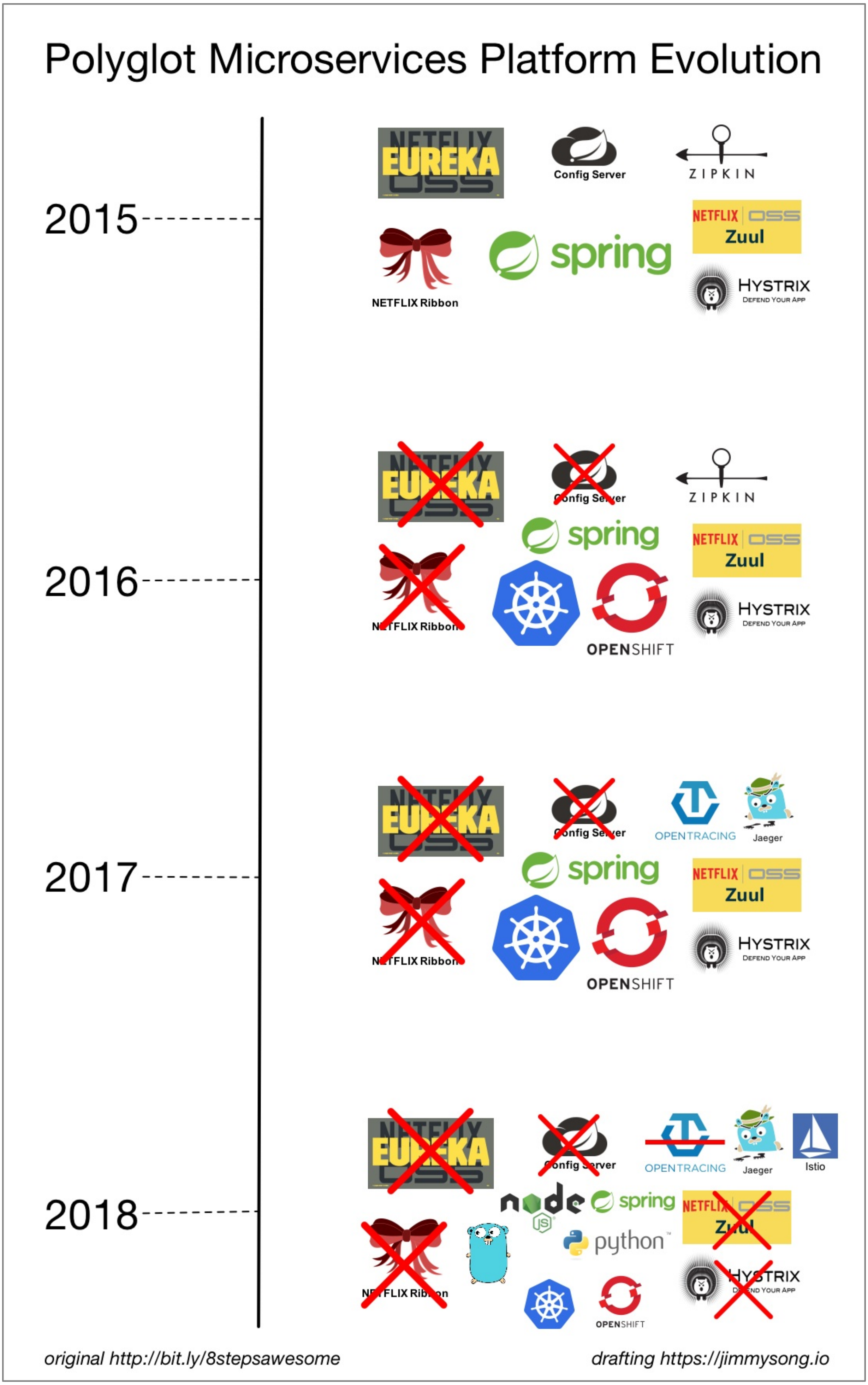
- AOP - Aspect Oriented Programming - spring.io

Service Mesh 服务网格

Service mesh 又译作 ”服务网格“，作为服务间通信的基础设施层。Buoyant 公司的 CEO Willian Morgan 在他的这篇文章 WHAT'S A SERVICE MESH? AND WHY DO I NEED ONE? 中解释了什么是 Service Mesh，为什么云原生应用需要 Service Mesh。

> A service mesh is a dedicated infrastructure layer for handling service-to-service communication. It's responsible for the reliable delivery of requests through the complex topology of services that comprise a modern, cloud native application. In practice, the service mesh is typically implemented as an array of lightweight network proxies that are deployed alongside application code, without the application needing to be aware.

今年来以 Istio 和 Linkderd 为代表的 Service Mesh 蓬勃发展，大有成为下一代语言异构微服务架构的王者之范，今天又碰巧看到了 Red Hat 的 Burr Sutter 提出了**8 Steps to Becoming Awesome with Kubernetes**，整个PPT一共60多页，很有建设性，点此跳转到我的GitHub上下载，我将其归档到cloud-native-slides-share中了。


Polyglot Microservices Platform Evolution
2015
2016
2017
2018
EUREKA
Config Server
ZIPKIN
NETFLIX Ribbon
spring
NETFLIX OSS
Zuul
HYSTRIX
DEFEND YOUR APP
OPENSHIFT
OPENTRACING
Jaeger
Istio
node
python
original http://bit.ly/8stepsawesome
drafting https://jimmysong.io

图片 - 下一代异构微服务架构

自我6月份初接触Istio依赖就发觉service mesh很好的解决了异构语言中的很多问题，而且是kuberentes service 上层不可或缺的服务间代理。关于istio的更多内容请参考 istio中文文档。

什么是 service mesh？

Service mesh 有如下几个特点：

- 应用程序间通讯的中间层
- 轻量级网络代理
- 应用程序无感知
- 解耦应用程序的重试/超时、监控、追踪和服务发现

目前两款流行的 service mesh 开源软件 Istio 和 Linkerd 都可以直接在 kubernetes 中集成，其中 Linkerd 已经成为 CNCF 成员。

理解 Service Mesh

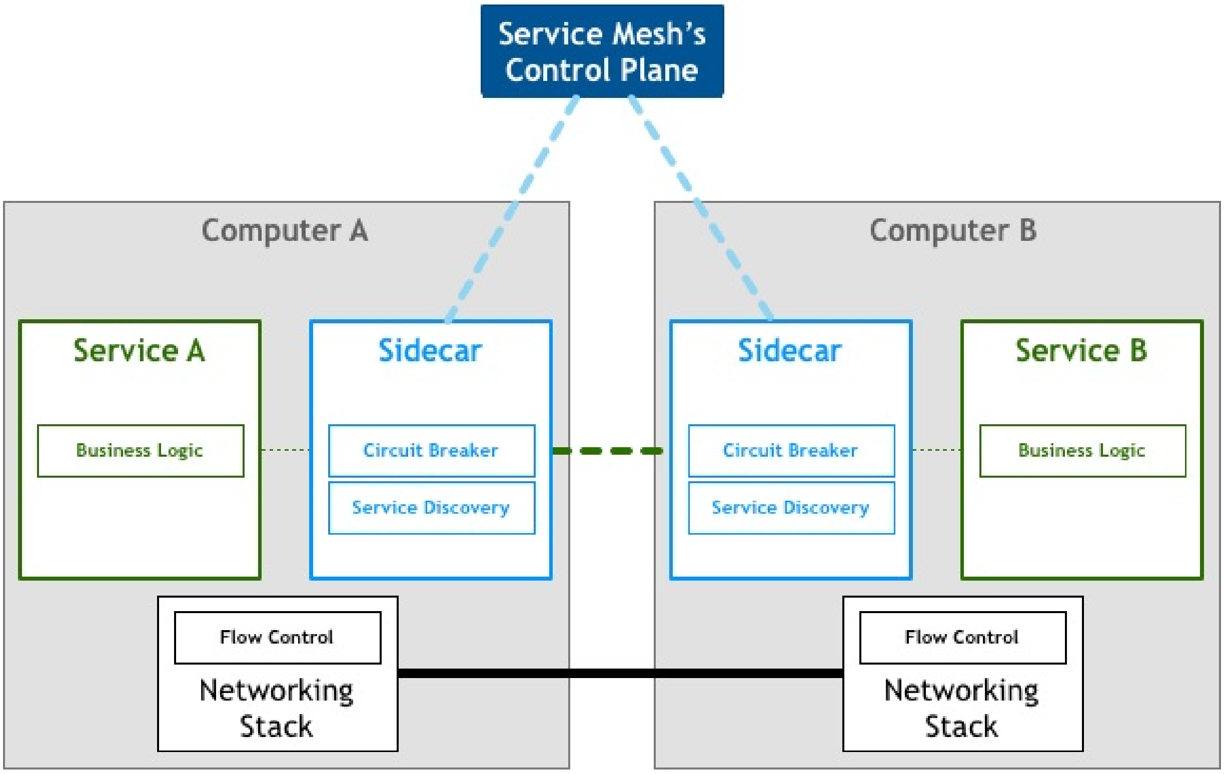
如果用一句话来解释什么是 Service Mesh，可以将它比作是应用程序或者说微服务间的 TCP/IP，负责服务之间的网络调用、限流、熔断和监控。对于编写应用程序来说一般无须关心 TCP/IP 这一层（比如通过 HTTP 协议的 RESTful 应用），同样使用 Service Mesh 也就无须关系服务之间的那些原来是通过应用程序或者其他框架实现的事情，比如 Spring Cloud、OSS，现在只要交给 Service Mesh 就可以了。

Phil Calçado 在他的这篇博客 Pattern: Service Mesh 中详细解释了 Service Mesh 的来龙去脉：

1. 从最原始的主机之间直接使用网线相连
2. 网络层的出现
3. 集成到应用程序内部的控制流
4. 分解到应用程序外部的控制流
5. 应用程序的中集成服务发现和断路器
6. 出现了专门用于服务发现和断路器的软件包/库，如 Twitter 的 Finagle 和

Facebook 的 Proxygen，这时候还是集成在应用程序内部
7. 出现了专门用于服务发现和断路器的开源软件，如 Netflix OSS、Airbnb 的 synapse 和 nerve
8. 最后作为微服务的中间层 service mesh 出现

Service mesh 的架构如下图所示：



图片 - *Service Mesh* 架构图

图片来自：Pattern: Service Mesh

Service mesh 作为 sidecar 运行，对应用程序来说是透明，所有应用程序间的流量都会通过它，所以对应用程序流量的控制都可以在 serivce mesh 中实现。

Service mesh如何工作？

下面以 Linkerd 为例讲解 service mesh 如何工作，Istio 作为 service mesh 的另一种实现原理与 linkerd 基本类似，后续文章将会详解 Istio 和 Linkerd 如何在 kubernetes 中工作。

1. Linkerd 将服务请求路由到目的地址，根据中的参数判断是到生产环境、测试环境还是 staging 环境中的服务（服务可能同时部署在这三个环境中），是路由到本地环境还是公有云环境？所有的这些路由信息可以动态配置，可以是全局配置也可以为某些服务单独配置。
2. 当 Linkerd 确认了目的地址后，将流量发送到相应服务发现端点，在 kubernetes 中是 service，然后 service 会将服务转发给后端的实例。
3. Linkerd 根据它观测到最近请求的延迟时间，选择出所有应用程序的实例中响应最快的实例。
4. Linkerd 将请求发送给该实例，同时记录响应类型和延迟数据。
5. 如果该实例挂了、不响应了或者进程不工作了，Linkerd 将把请求发送到其他实例上重试。
6. 如果该实例持续返回 error，Linkerd 会将该实例从负载均衡池中移除，稍后再周期性得重试。
7. 如果请求的截止时间已过，Linkerd 主动失败该请求，而不是再次尝试添加负载。
8. Linkerd 以 metric 和分布式追踪的形式捕获上述行为的各个方面，这些追踪信息将发送到集中 metric 系统。

为何使用 service mesh？

Service mesh 并没有给我们带来新功能，它是用于解决其他工具已经解决过的问题，只不过这次是在 Cloud Native 的 kubernetes 环境下的实现。

在传统的 MVC 三层 Web 应用程序架构下，服务之间的通讯并不复杂，在应用程序内部自己管理即可，但是在现今的复杂的大型网站情况下，单体应用被分解为众多的微服务，服务之间的依赖和通讯十分复杂，出现了 twitter 开发的 Finagle、Netflix 开发的 Hystrix 和 Google 的 Stubby 这样的 ”胖客户端“ 库，这些就是早期的 service mesh，但是它们都近适用于特定的环境和特定的开发语言，并不能作为平台级的 service mesh 支持。

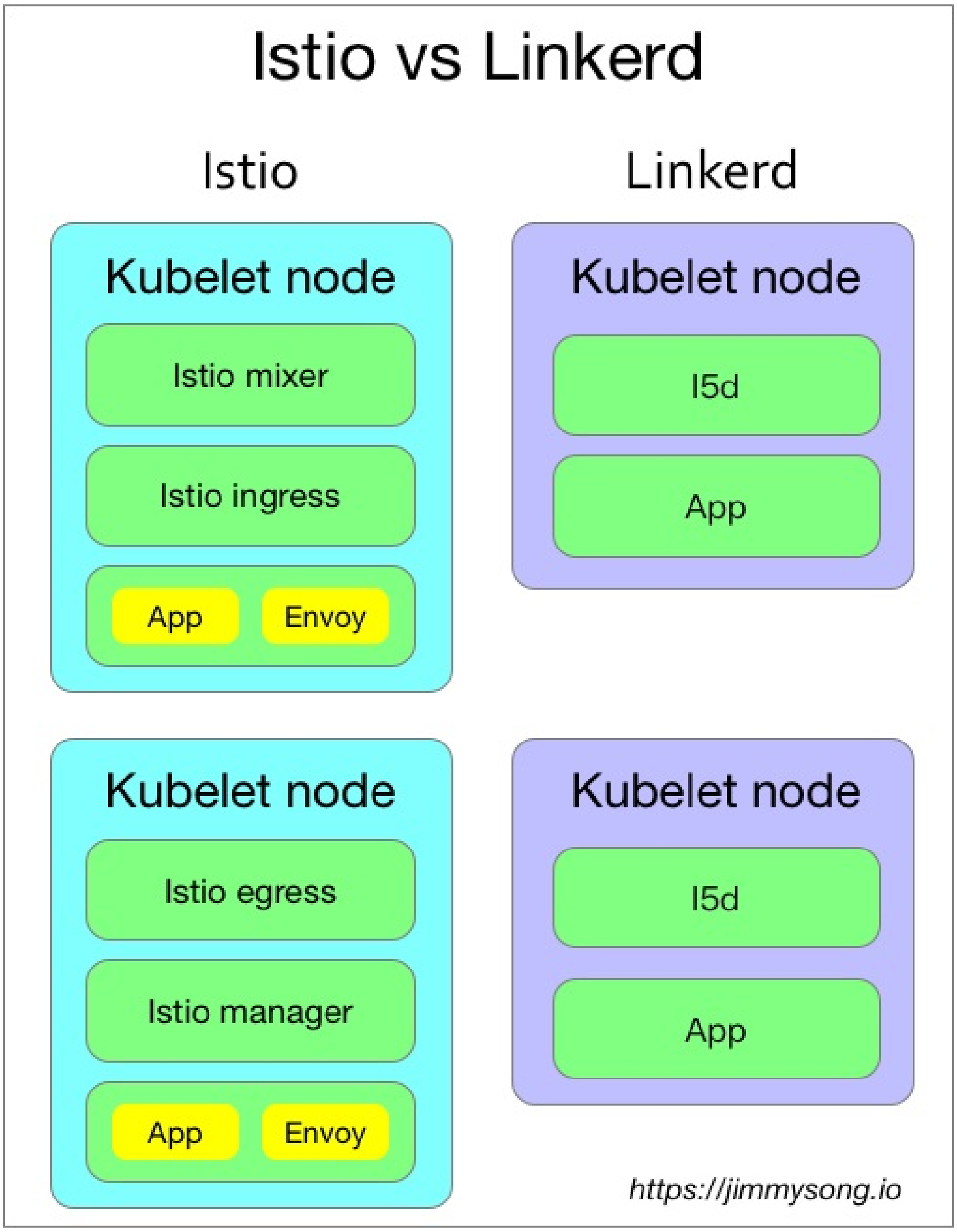
在 Cloud Native 架构下，容器的使用给予了异构应用程序的更多可行性，kubernetes 增强的应用的横向扩容能力，用户可以快速的编排出复杂环境、复杂依赖关系的应用程序，同时开发者又无须过分关心应用程序的监控、扩展性、服务发现和分布式追踪这些繁琐的事情而专注于程序开发，赋予开发者更多的创造性。

Istio VS Linkerd

当前的Service Mesh实现主要有两大阵营，要给是Linkerd（也是最初提出该概念的），另一个是Istio，当然还有很多其他号称也是Service Mesh，比如Nginx出品的Nginmesh。

Feature	Istio	Linkerd
部署架构	Envoy/Sidecar	DaemonSets
易用性	复杂	简单
支持平台	kuberentes	kubernetes/mesos/Istio/local
当前版本	0.3.0	1.3.3
是否已有生产部署	否	是

下图是Istio和Linkerd架构的不同，Istio是使用Sidecar模式，将Envoy植入到Pod中，而Linkerd则是在每台node上都以DaemonSet的方式运行。



图片 - *Istio vs linkerd*

关于Istio和Linkerd的详细信息请参考 安装并试用Istio service mesh 与 Linkerd 使用指南。

另外出品Linkerd的公司buoyant又推出了conduit，这是一种更轻量级的Service Mesh。

参考

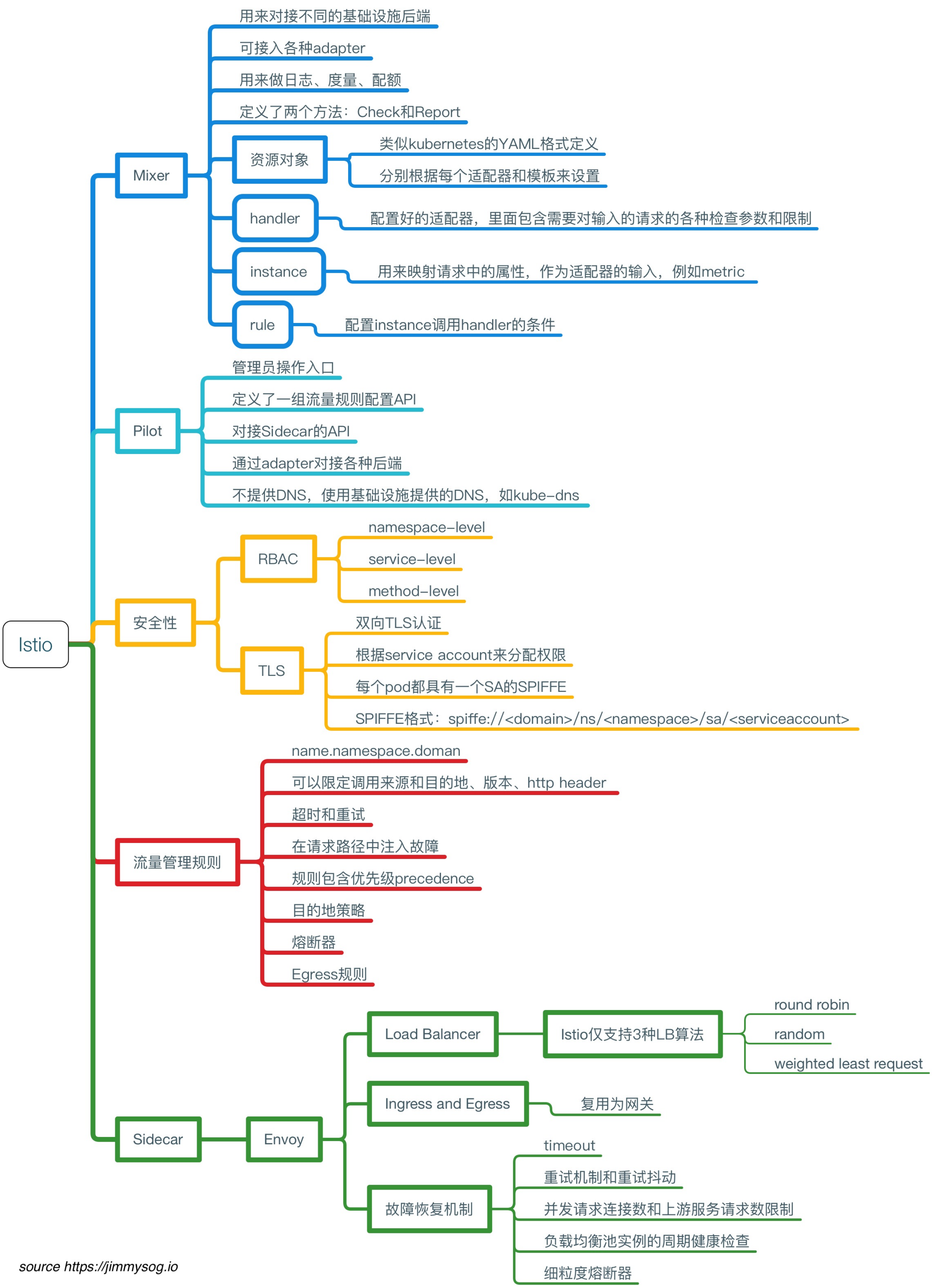
- WHAT'S A SERVICE MESH? AND WHY DO I NEED ONE?

- So what even is a Service Mesh? Hot take on Istio and Linkerd
- linkerd: A service mesh for AWS ECS
- Introducing Istio: A robust service mesh for microservices
- Application Network Functions With ESBs, API Management, and Now.. Service Mesh?
- Pattern: Service Mesh
- Istio官方文档中文版

Istio简介

Istio是由Google、IBM和Lyft开源的微服务管理、保护和监控框架。Istio为希腊语，意思是”起航“。关于Istio的详细信息请参考Istio官方文档和Istio中文文档。

TL;DR 关于Istio中的各个组件和一些关键信息请参考下面的mindmap。



图片 - *Istio的mindmap*

简介

使用istio可以很简单的创建具有负载均衡、服务间认证、监控等功能的服务网络，而不需要对服务的代码进行任何修改。你只需要在部署环境中，例如Kubernetes的pod里注入一个特别的sidecar proxy来增加对istio的支持，用来截获微服务之间的网络流量。

目前版本的istio只支持kubernetes，未来计划支持其他其他环境。

另外，Istio的前身是IBM开源的Amalgam8，追本溯源，我们来看下它的特性。

Amalgam8

Amalgam8的网站上说，它是一个**Content-based Routing Fabric for Polyglot Microservices**，简单、强大且开源。

Amalgam8是一款基于内容和版本的路由布局，用于集成多语言异构体微服务。 其control plane API可用于动态编程规则，用于在正在运行的应用程序中跨微服务进行路由和操作请求。

以内容/版本感知方式路由请求的能力简化了DevOps任务，如金丝雀和红/黑发布，A/B Test和系统地测试弹性微服务。

可以使用Amalgam8平台与受欢迎的容器运行时（如Docker，Kubernetes，Marathon / Mesos）或其他云计算提供商（如IBM Bluemix，Google Cloud Platform或Amazon AWS）。

特性

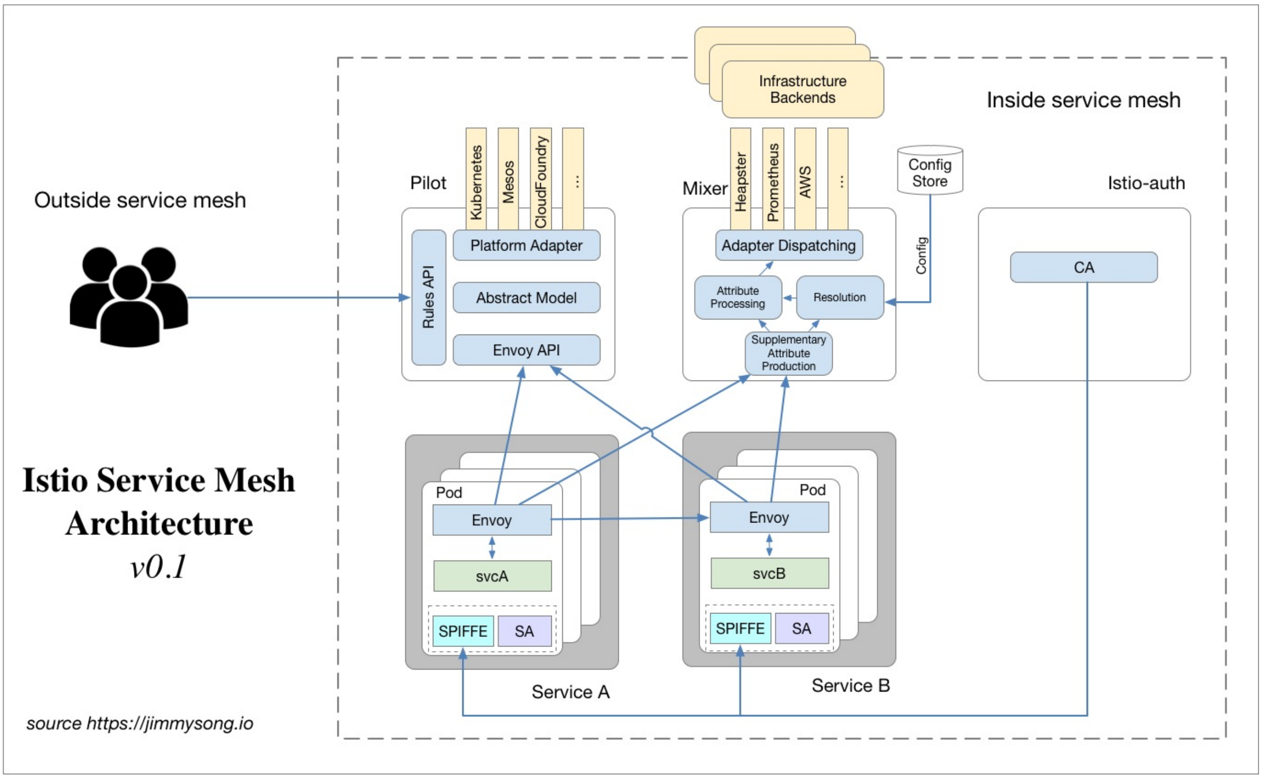
使用istio的进行微服务管理有如下特性：

- 流量管理：控制服务间的流量和API调用流，使调用更可靠，增强不同环境下的网络鲁棒性。
- 可观测性：了解服务之间的依赖关系和它们之间的性质和流量，提供快速识别定位问题的能力。
- 策略实施：通过配置mesh而不是以改变代码的方式来控制服务之间的访问策略。
- 服务识别和安全：提供在mesh里的服务可识别性和安全性保护。

未来将支持多种平台，不论是kubernetes、Mesos、还是云。同时可以集成已有的ACL、日志、监控、配额、审计等。

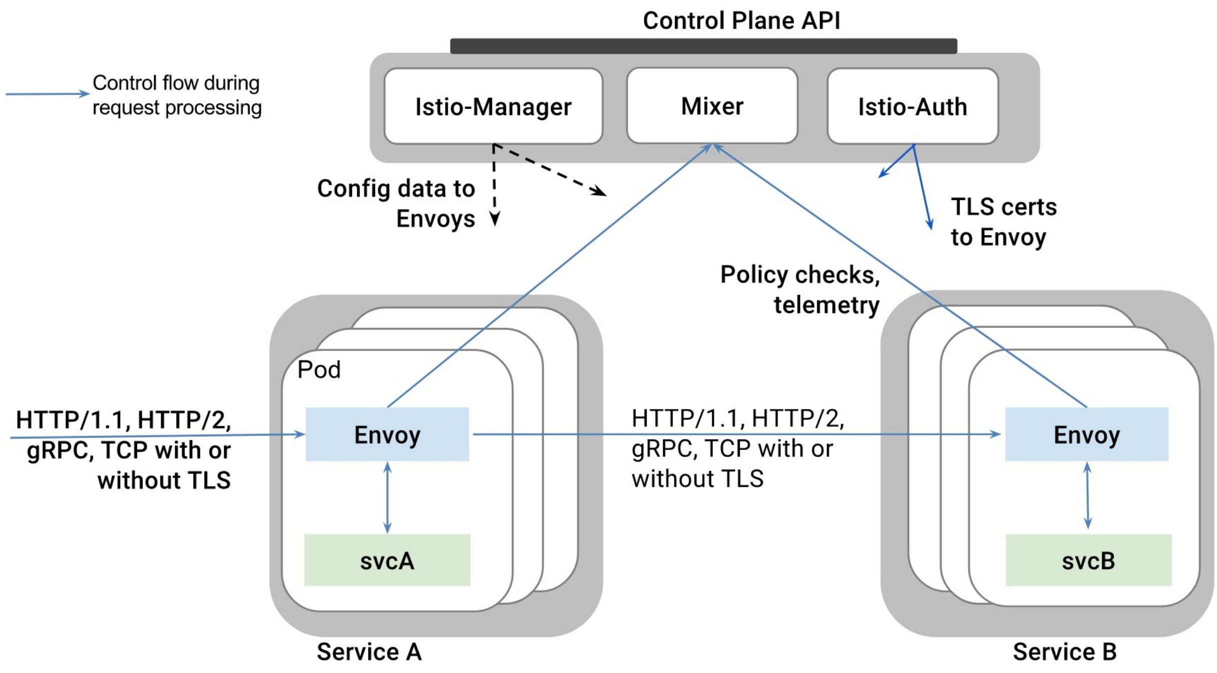
架构

下面是Istio的架构图。



图片 - *Istio*架构图

下图是Istio中控制平面与数据平面的交互流程图。



图片 - *Istio*的控制平面和数据平面

Istio架构分为控制平面和数据平面。

- 数据平面：由一组智能代理（Envoy）作为sidecar部署，协调和控制所有microservices之间的网络通信。
- 控制平面：负责管理和配置代理路由流量，以及在运行时执行的政策。

Envoy

Istio使用Envoy代理的扩展版本，该代理是以C++开发的高性能代理，用于调解service mesh中所有服务的所有入站和出站流量。 Istio利用了Envoy的许多内置功能，例如动态服务发现，负载平衡，TLS终止，HTTP/2&gRPC代理，断路器，运行状况检查，基于百分比的流量拆分分阶段上线，故障注入和丰富指标。

Envoy在kubernetes中作为pod的sidecar来部署。 这允许Istio将大量关于流量行为的信号作为属性提取出来，这些属性又可以在Mixer中用于执行策略决策，并发送给监控系统以提供有关整个mesh的行为的信息。 Sidecar代理模型还允许你将Istio功能添加到现有部署中，无需重新构建或重写代码。 更多信息参见设计目标。

Mixer

Mixer负责在service mesh上执行访问控制和使用策略，并收集Envoy代理和其他服务的遥测数据。代理提取请求级属性，发送到mixer进行评估。有关此属性提取和策略评估的更多信息，请参见Mixer配置。 混音器包括一个灵活的插件模型，使其能够与各种主机环境和基础架构后端进行接口，从这些细节中抽象出Envoy代理和Istio管理的服务。

Istio Manager

Istio-Manager用作用户和Istio之间的接口，收集和验证配置，并将其传播到各种Istio组件。它从Mixer和Envoy中抽取环境特定的实现细节，为他们提供独立于底层平台的用户服务的抽象表示。 此外，流量管理规则（即通用4层规则和七层HTTP/gRPC路由规则）可以在运行时通过Istio-Manager进行编程。

Istio-auth

Istio-Auth提供强大的服务间和最终用户认证，使用相互TLS，内置身份和凭据管理。它可用于升级service mesh中的未加密流量，并为运营商提供基于服务身份而不是网络控制的策略的能力。 Istio的未来版本将增加细粒度的访问控制和审计，以使用各种访问控制机制（包括属性和基于角色的访问控制以及授权hook）来控制和监控访问你服务、API或资源的人员。

参考

- Istio开源平台发布，Google、IBM和Lyft分别承担什么角色？
- Istio：用于微服务的服务啮合层
- Istio Overview

安装并试用Istio service mesh

官方文档地址 快速开始

本文根据官网的文档整理而成，步骤包括安装**istio 0.5.1**并创建一个bookinfo的微服务来测试istio的功能。

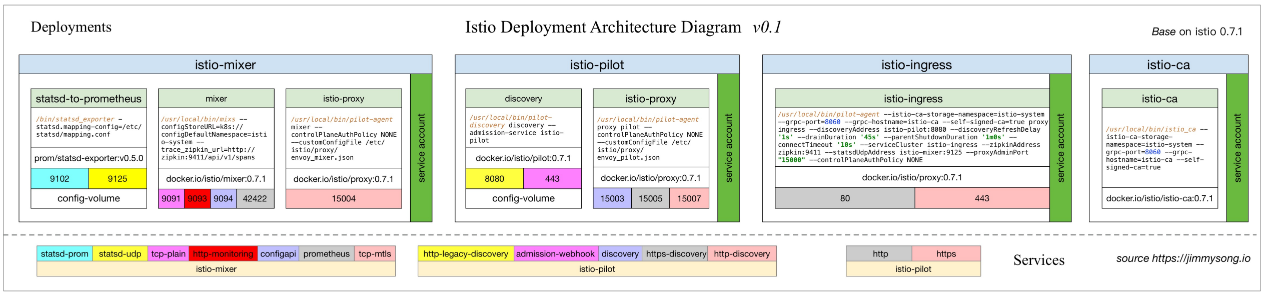
文中使用的yaml文件可以在kubernetes-handbook的 `manifests/istio` 目录中找到，如果镜像pull失败，请根据官网的镜像自行修改。

安装环境

- CentOS 7.4.1708
- Docker 17.12.0-ce
- Kubernetes 1.8.5

部署结构

Istio 的控制平面部署在 Kubernetes 中的部署架构如下图所示。



图片 - *Istio 在 Kubernetes 中的部署架构图*

我们可以清楚的看到 Istio 控制平面的几个组件的部署运行的命令与开发的端口，以及端口与服务之间的映射的关系。

安装

1.下载安装包

下载地址：https://github.com/istio/istio/releases

下载Linux版本的当前最新版安装包

```
wget https://github.com/istio/istio/releases/download/0.5.1/isti
o-0.5.1-linux.tar.gz
```

2.解压

解压后，得到的目录结构如下：

```
├── bin
│   └── istioctl
├── install
│   ├── ansible
│   ├── consul
│   ├── eureka
│   ├── gcp
│   ├── kubernetes
│   ├── README.md
│   └── tools
├── istio.VERSION
├── LICENSE
├── README.md
├── samples
│   ├── bookinfo
│   ├── CONFIG-MIGRATION.md
│   ├── helloworld
│   ├── httpbin
│   ├── kubernetes-blog
│   ├── rawvm
│   ├── README.md
│   └── sleep
└── tools
    ├── cache_buster.yaml
    ├── deb
    ├── githubContrib
    ├── minikube.md
    ├── perf_istio_rules.yaml
    ├── perf_k8svcs.yaml
    ├── README.md
    ├── rules.yml
    ├── setup_perf_cluster.sh
    ├── setup_run
    ├── update_all
    └── vagrant
```

从文件里表中可以看到，安装包中包括了kubernetes的yaml文件，示例应用和安装模板。

3.先决条件

以下说明要求您可以访问启用了RBAC（基于角色的访问控制）的Kubernetes1.7.3或更新的群集。您还需要安装1.7.3或更高版本。如果您希望启用automatic sidecar injection，则需要Kubernetes 1.9或更高版本。kubectl

注意：如果您安装了Istio 0.1.x，请在安装新版本之前彻底卸载它（包括适用于所有启用Istio的应用程序窗口的Istio支架）。 安装或升级Kubernetes CLIkubectl以匹配群集支持的版本（CRD支持版本为1.7或更高版本）。

根据您的Kubernetes提供者：

要在本地安装Istio，请安装最新版本的Minikube（版本0.22.1或更高版本）。

4.安装步骤

从0.2版本开始，Istio安装在它自己的istio-system命名空间中，并且可以管理来自所有其他命名空间的服务。

转至Istio发布页面以下载与您的操作系统相对应的安装文件。如果您使用的是MacOS或Linux系统，以下命令自动下载并提取最新版本：

```
curl -L https://git.io/getLatestIstio | sh -
```

解压缩安装文件并将目录更改为文件位置。

安装目录包含：

```
Installation .yaml Kubernetes的安装文件
Sample/ 示例应用程序
bin/istioctl 二进制bin/文件 在手动注入Envoy作为附属代理并创建路由规则和策略时使用.
istio.VERSION配置文件
```

例如，如果包是istio-0.5（初步）

```
cd istio-0.5 (preliminary)
```

将istioctl客户端添加到您的PATH。例如，在MacOS或Linux系统上运行以下命令：

```
export PATH=$PWD/bin:$PATH
```

安装Istio的核心组件。从下面两个互相排斥的选项中选择一个，或者用Helm Chart交替安装： a）安装Istio而不启用侧车间的相互TLS认证。为具有现有应用程序的群集，使用Istio辅助车的服务需要能够与其他非Istio Kubernetes服务以及使用活动性和准备就绪探测器，无头服务或StatefulSets的应用程序通信的应用程序选择此选项。

```
kubectl apply -f install/kubernetes/istio.yaml
```

要么

b）安装Istio并启用侧柜之间的相互TLS认证：

```
kubectl apply -f install/kubernetes/istio-auth.yaml
```

这两个选项都会创建istio-system命名空间以及所需的RBAC权限，并部署Istio-Pilot，Istio-Mixer，Istio-Ingress和Istio-CA（证书颁发机构）。

可选：如果您的群集的Kubernetes版本是1.9或更高，并且您希望启用自动代理注入，请安装sidecar injector webhook。 验证安装 请确保以下Kubernetes服务部署：istio-pilot，istio-mixer，istio-ingress。

```
kubectl get svc -n istio-system
```

```
NAME            CLUSTER-IP      EXTERNAL-IP      PORT(S)
                AGE
istio-ingress   10.83.245.171   35.184.245.62    80:32730/TCP,4
43:30574/TCP    5h
istio-pilot     10.83.251.173   <none>           8080/TCP,8081/
TCP             5h
istio-mixer     10.83.244.253   <none>           9091/TCP,9094/
TCP,42422/TCP   5h
```

> 注意：如果您的集群中不支持外部负载平衡（例如，Minikube）的环境中运行，该external-ip的istio-ingress会显示。您必须使用NodePort访问应用程序，或使用端口转发。

修改istio.yaml中的istio-ingress service的type为ClusterIP，并设置 `nodePort` ，默认为32000。

确保相应Kubernetes容器都运行起来： `istio-pilot-*` 、 `istio-mixer-*` 、 `istio-ingress-*` 、 `istio-ca-*` ，和可选的 `istio-sidecar-injector-*` 。

```
kubectl get pods -n istio-system
```

```
istio-ca-3657790228-j21b9                   1/1       Running   0
        5h
istio-ingress-1842462111-j3vcs              1/1       Running   0
        5h
istio-sidecar-injector-184129454-zdgf5      1/1       Running   0
        5h
istio-pilot-2275554717-93c43                1/1       Running   0
        5h
istio-mixer-2104784889-20rm8                2/2       Running   0
        5h
```

部署您的应用程序

您现在可以部署您自己的应用程序或者像Bookinfo一样随安装提供的示例应用程序之一。注意：应用程序必须对所有HTTP通信使用HTTP/1.1或HTTP/2.0协议，因为HTTP/1.0不受支持。

如果您启动了Istio-sidecar-injector，如上所示，您可以直接使用应用程序部署应用程序kubectl create。

Istio Sidecar注入器会自动将Envoy容器注入到您的应用程序窗格中，假设运行在标有名称空间的名称空间中 `istio-injection=enabled`

```
kubectl label namespace <namespace> istio-injection=enabled
```

```
kubectl create -n <namspace> -f <your-app-spec>.yaml
```

如果您没有安装Istio-sidecar-injector，则在部署它们之前，必须使用istioctl kube-inject将Envoy容器手动注入应用程序窗格中：

```
kubectl create -f <(istioctl kube-inject -f <your-app-spec>.yaml)
```

卸载

卸载Istio sidecar进样器：

如果您启用Istio-sidecar-injector，请将其卸载：

```
kubectl delete -f install/kubernetes/istio-sidecar-injector-with
-ca-bundle.yaml
```

卸载Istio核心组件。对于0.6（初始）发行版，卸载将删除RBAC权限，istio-system命名空间和分层下的所有资源。忽略不存在资源的错误是安全的，因为它们可能已被分层删除。

a）如果您在禁用相互TLS身份验证的情况下安装了Istio：

```
kubectl delete -f install/kubernetes/istio.yaml
```

要么

b）如果您在启用相互TLS身份验证的情况下安装了Istio：

```
kubectl delete -f install/kubernetes/istio-auth.yaml
```

7.安装监控插件

安装插件

```
kubectl apply -f install/kubernetes/addons/prometheus.yaml
kubectl apply -f install/kubernetes/addons/grafana.yaml
kubectl apply -f install/kubernetes/addons/servicegraph.yaml
kubectl apply -f install/kubernetes/addons/zipkin.yaml
```

在traefik ingress中增加增加以上几个服务的配置，同时增加istio-ingress配置。

```yaml
  - host: grafana.istio.io
    http:
      paths:
      - path: /
        backend:
          serviceName: grafana
          servicePort: 3000
  - host: servicegraph.istio.io
    http:
      paths:
      - path: /
        backend:
          serviceName: servicegraph
          servicePort: 8088
  - host: prometheus.istio.io
    http:
      paths:
      - path: /
        backend:
          serviceName: prometheus
          servicePort: 9090
  - host: zipkin.istio.io
    http:
      paths:
      - path: /
        backend:
          serviceName: zipkin
          servicePort: 9411
  - host: ingress.istio.io
    http:
      paths:
      - path: /
        backend:
          serviceName: istio-ingress
          servicePort: 80
```

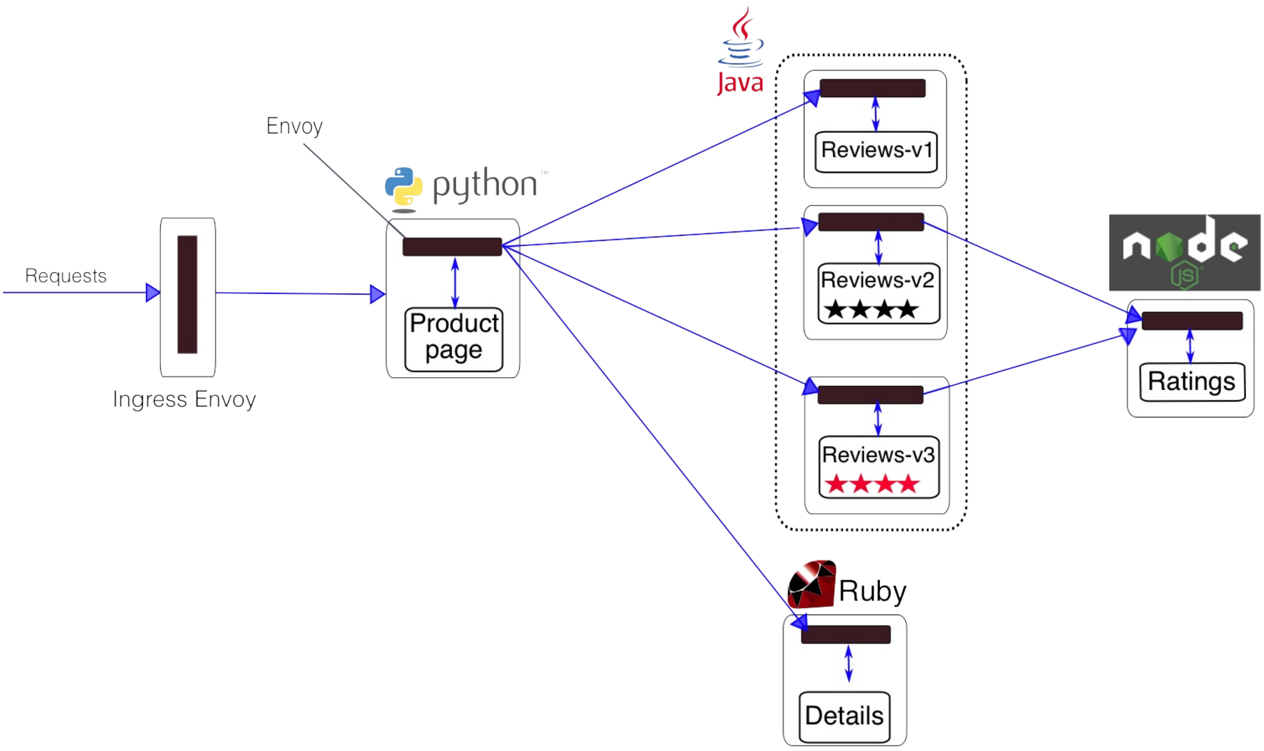
测试

我们使用Istio提供的测试应用bookinfo微服务来进行测试。

该微服务用到的镜像有：

```
istio/examples-bookinfo-details-v1
istio/examples-bookinfo-ratings-v1
istio/examples-bookinfo-reviews-v1
istio/examples-bookinfo-reviews-v2
istio/examples-bookinfo-reviews-v3
istio/examples-bookinfo-productpage-v1
```

该应用架构图如下：



图片 - *BookInfo Sample*应用架构图

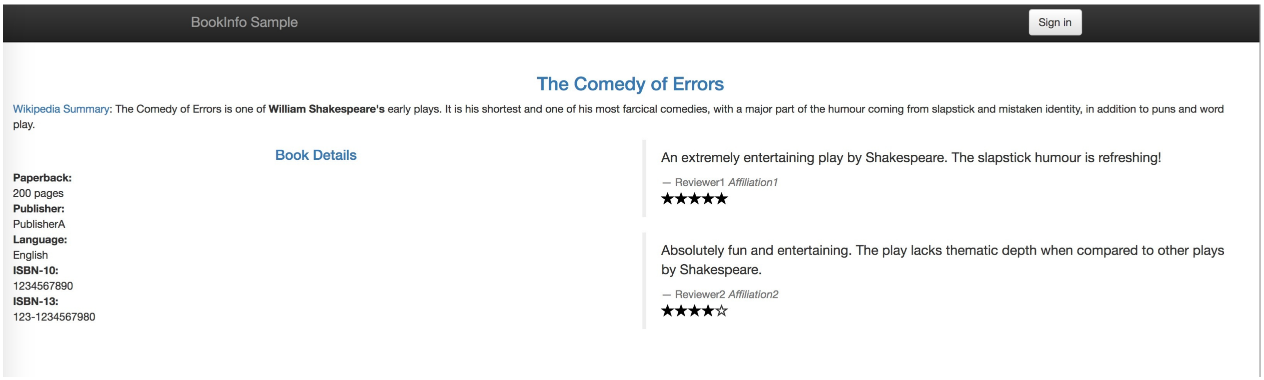
部署应用

```
kubectl create -f <(istioctl kube-inject -f samples/apps/bookinf
o/bookinfo.yaml)
```

`Istio kube-inject` 命令会在 `bookinfo.yaml` 文件中增加Envoy sidecar信息。参考：https://istio.io/docs/reference/commands/istioctl.html#istioctl-kube-inject

在本机的 `/etc/hosts` 下增加VIP节点和 `ingress.istio.io` 的对应信息，具体步骤参考：边缘节点配置，或者使用gateway ingress来访问服务，

如果将 `productpage` 配置在了ingress里了，那么在浏览器中访问 http://ingress.istio.io/productpage，如果使用了istio默认的 `gateway` ingress配置的话，ingress service使用 `nodePort` 方式暴露的默认使用32000端口，那么可以使用http://任意节点的IP:32000/productpage来访问。



图片 - *BookInfo Sample*页面

多次刷新页面，你会发现有的页面上的评论里有星级打分，有的页面就没有，这是因为我们部署了三个版本的应用，有的应用里包含了评分，有的没有。Istio根据默认策略随机将流量分配到三个版本的应用上。

查看部署的bookinfo应用中的 `productpage-v1` service和deployment，查看 `productpage-v1` 的pod的详细json信息可以看到这样的结构：

```
$ kubectl get pod productpage-v1-944450470-bd530 -o json
```

见productpage-v1-istio.json文件。从详细输出中可以看到这个Pod中实际有两个容器，这里面包括了 `initContainer`，作为istio植入到kubernetes deployment中的sidecar。

```
"initContainers": [
            {
                "args": [
                    "-p",
                    "15001",
                    "-u",
                    "1337"
                ],
                "image": "docker.io/istio/init:0.1",
                "imagePullPolicy": "Always",
                "name": "init",
                "resources": {},
```

```
                "securityContext": {
                    "capabilities": {
                        "add": [
                            "NET_ADMIN"
                        ]
                    }
                },
                "terminationMessagePath": "/dev/termination-log",

                "terminationMessagePolicy": "File",
                "volumeMounts": [
                    {
                        "mountPath": "/var/run/secrets/kubernete
s.io/serviceaccount",
                        "name": "default-token-3l9f0",
                        "readOnly": true
                    }
                ]
            },
            {
                "args": [
                    "-c",
                    "sysctl -w kernel.core_pattern=/tmp/core.%e.
%p.%t \u0026\u0026 ulimit -c unlimited"
                ],
                "command": [
                    "/bin/sh"
                ],
                "image": "alpine",
                "imagePullPolicy": "Always",
                "name": "enable-core-dump",
                "resources": {},
                "securityContext": {
                    "privileged": true
                },
                "terminationMessagePath": "/dev/termination-log",

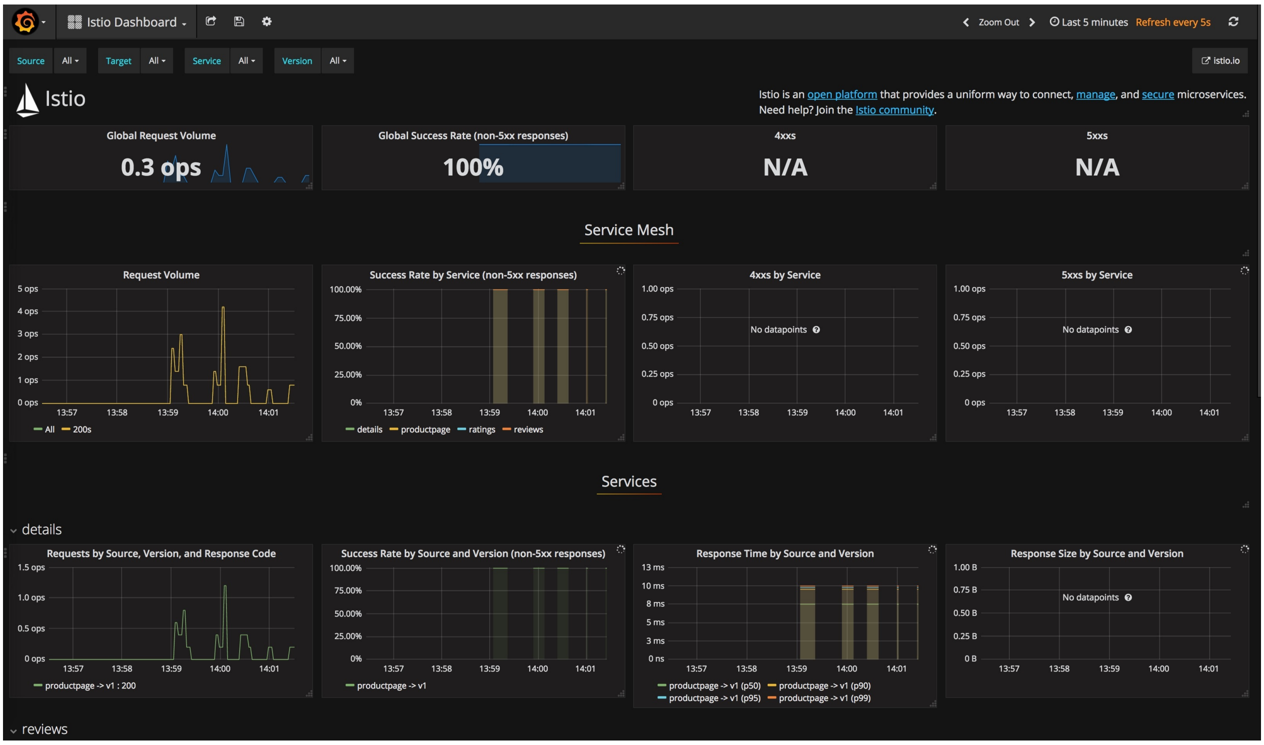
                "terminationMessagePolicy": "File",
                "volumeMounts": [
                    {
                        "mountPath": "/var/run/secrets/kubernete
s.io/serviceaccount",
                        "name": "default-token-3l9f0",
                        "readOnly": true
                    }
                ]
            }
        ],
```

监控

不断刷新productpage页面，将可以在以下几个监控中看到如下界面。

Grafana页面

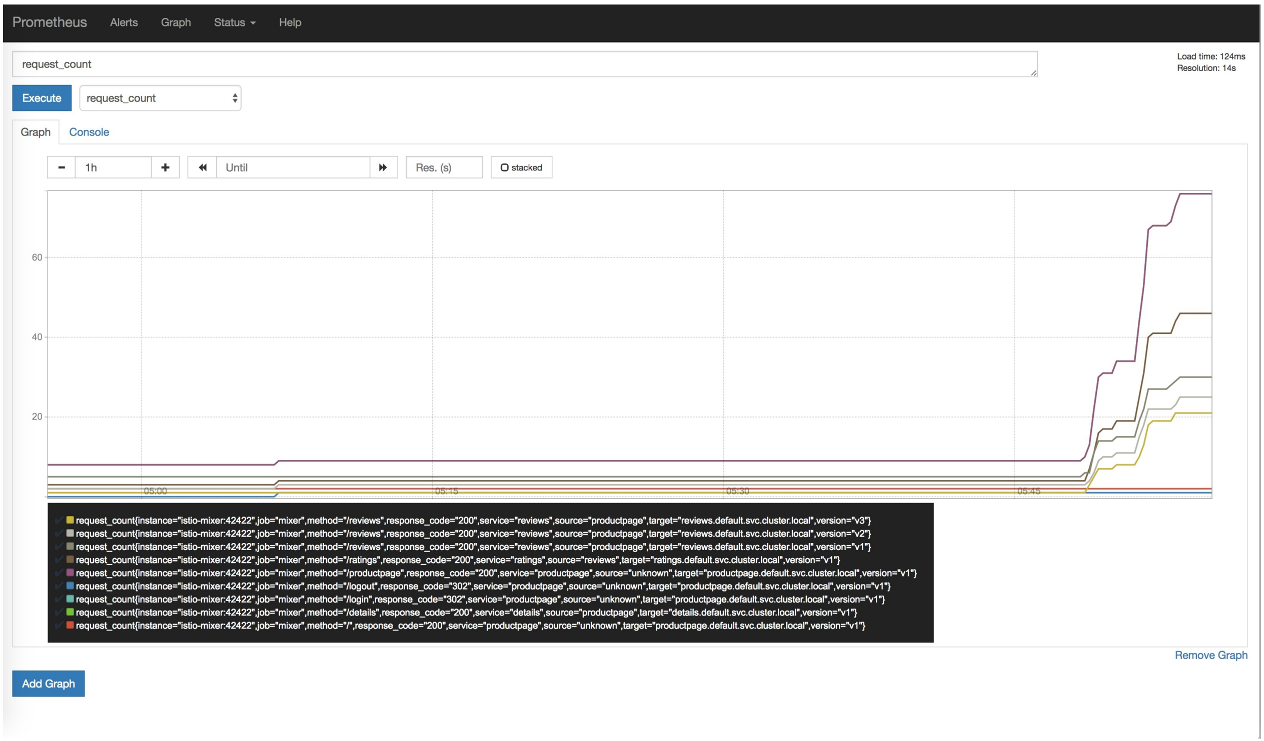
http://grafana.istio.io



图片 - *Istio Grafana*界面

Prometheus页面

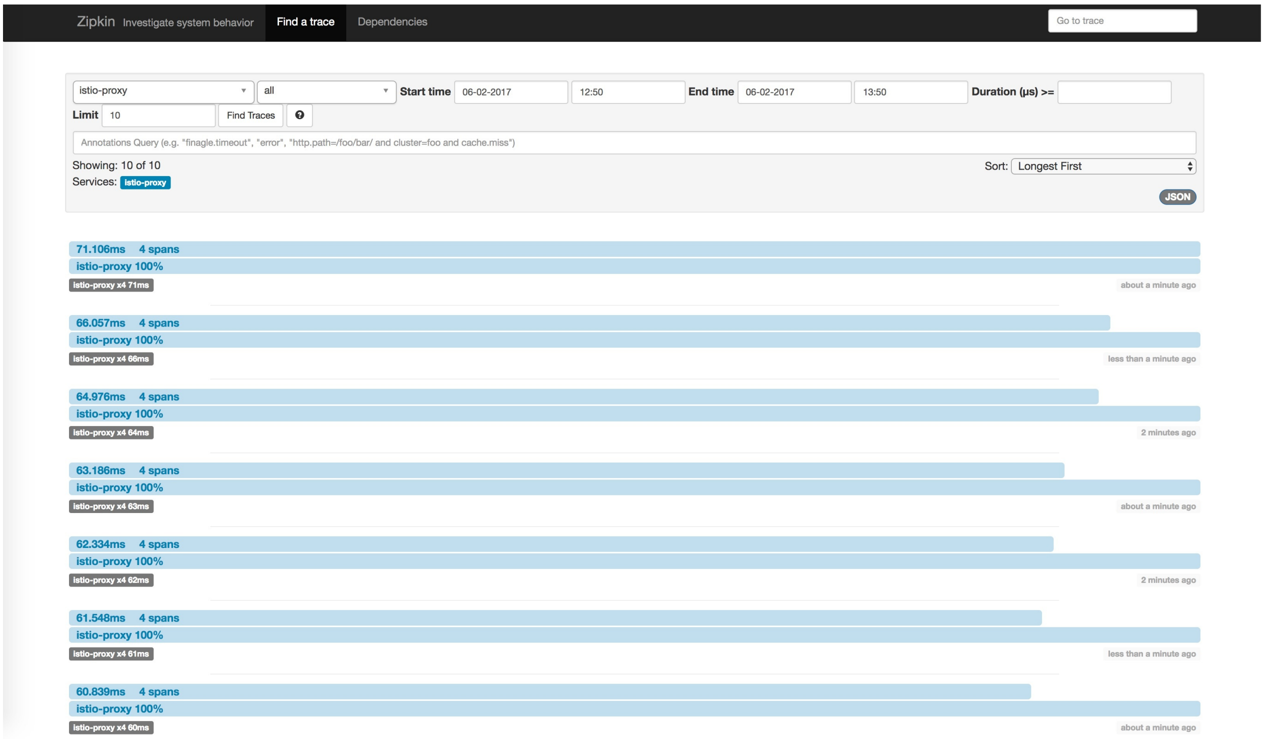
http://prometheus.istio.io



图片 - *Prometheus*页面

Zipkin页面

http://zipkin.istio.io



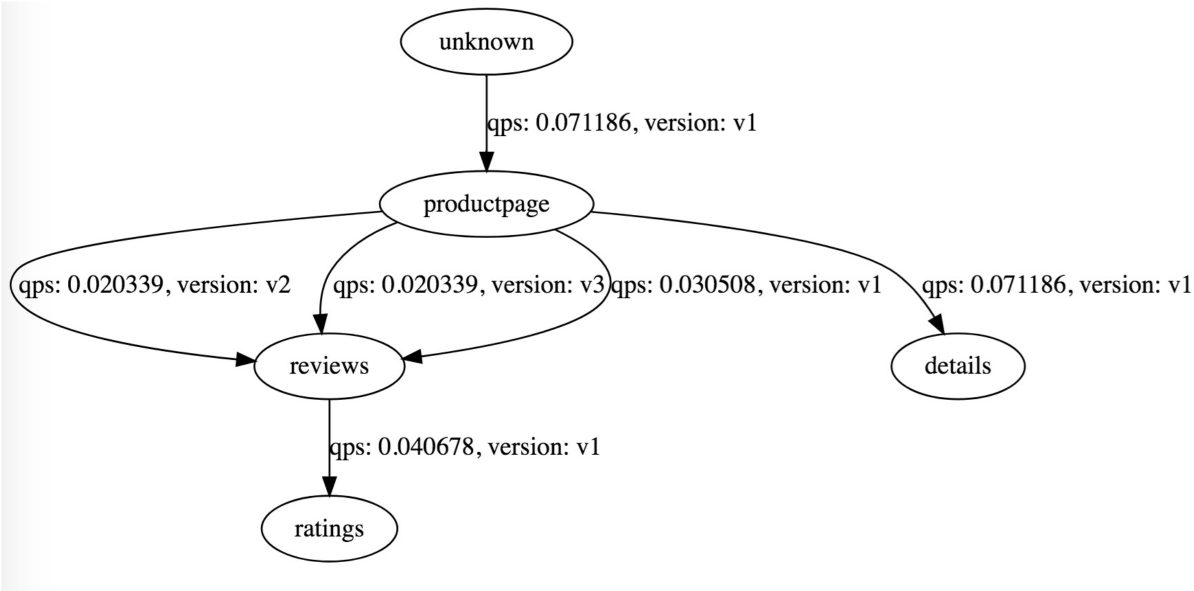
图片 - *Zipkin*页面

ServiceGraph页面

http://servicegraph.istio.io/dotviz

可以用来查看服务间的依赖关系。

访问 http://servicegraph.istio.io/graph 可以获得json格式的返回结果。



图片 - *ServiceGraph*页面

更进一步

BookInfo示例中有三个版本的 `reviews` ，可以使用istio来配置路由请求，将流量分发到不同版本的应用上。参考[Configuring Request Routing](Configuring Request Routing)。

还有一些更高级的功能，我们后续将进一步探索。

参考

- Installing Istio
- BookInfo sample

配置请求的路由规则

在上一节安装istio中我们创建BookInfo的示例，熟悉了Istio的基本功能，现在我们再来看一下istio的高级特性——配置请求的路由规则。

使用istio我们可以根据权重和**HTTP headers**来动态配置请求路由。

基于内容的路由

因为BookInfo示例部署了3个版本的评论微服务，我们需要设置一个默认路由。 否则，当你多次访问应用程序时，会注意到有时输出包含星级，有时候又没有。 这是因为没有明确的默认版本集，Istio将以随机方式将请求路由到服务的所有可用版本。

注意：假定您尚未设置任何路由。如果您已经为示例创建了冲突的路由规则，则需要在以下命令中使用replace而不是create。

下面这个例子能够根据网站的不同登陆用户，将流量划分到服务的不同版本和实例。跟kubernetes中的应用一样，所有的路由规则都是通过声明式的yaml配置。关于 `reviews:v1` 和 `reviews:v2` 的唯一区别是，v1没有调用评分服务，productpage页面上不会显示评分星标。

1. 将微服务的默认版本设置成v1。

   ```
   istioctl create -f samples/apps/bookinfo/route-rule-all-v1.yaml
   ```

   使用以下命令查看定义的路由规则。

   ```
   istioctl get route-rules -o yaml
   ```

```yaml
type: route-rule
name: details-default
namespace: default
spec:
destination: details.default.svc.cluster.local
precedence: 1
route:
- tags:
    version: v1
---
type: route-rule
name: productpage-default
namespace: default
spec:
destination: productpage.default.svc.cluster.local
precedence: 1
route:
- tags:
    version: v1
---
type: route-rule
name: reviews-default
namespace: default
spec:
destination: reviews.default.svc.cluster.local
precedence: 1
route:
- tags:
    version: v1
---
type: route-rule
name: ratings-default
namespace: default
spec:
destination: ratings.default.svc.cluster.local
precedence: 1
route:
- tags:
    version: v1
---
```

由于对代理的规则传播是异步的，因此在尝试访问应用程序之前，需要等待几秒钟才能将规则传播到所有pod。

2. 在浏览器中打开BookInfo URL（http://$GATEWAY_URL/productpage ，我们在上一节中使用的是 http://ingress.istio.io/productpage ）您应该会看到BookInfo应用程序的产品页面显示。 注意，产品页面上没有评分星，因为 `reviews:v1` 不访问评级服务。

3. 将特定用户路由到 `reviews:v2` 。

   为测试用户jason启用评分服务，将productpage的流量路由到 `reviews:v2` 实例上。

   ```
   istioctl create -f samples/apps/bookinfo/route-rule-reviews-test-v2.yaml
   ```

   确认规则生效：

   ```
   istioctl get route-rule reviews-test-v2
   ```

   ```
   destination: reviews.default.svc.cluster.local
   match:
     httpHeaders:
       cookie:
         regex: ^(.*?;)?(user=jason)(;.*)?$
   precedence: 2
   route:
   - tags:
       version: v2
   ```

4. 使用jason用户登陆productpage页面。

   你可以看到每个刷新页面时，页面上都有一个1到5颗星的评级。如果你使用其他用户登陆的话，将因继续使用 `reviews:v1` 而看不到星标评分。

内部实现

在这个例子中，一开始使用istio将100%的流量发送到BookInfo服务的 `reviews:v1` 的实例上。然后又根据请求的header（例如用户的cookie）将流量选择性的发送到 `reviews:v2` 实例上。

验证了v2实例的功能后，就可以将全部用户的流量发送到v2实例上，或者逐步的迁移流量，如10%、20%直到100%。

如果你看了故障注入这一节，你会发现v2版本中有个bug，而在v3版本中修复了，你想将流量迁移到 `reviews:v1` 迁移到 `reviews:v3` 版本上，只需要运行如下命令：

1. 将50%的流量从 `reviews:v1` 转移到 `reviews:v3` 上。

   ```
   istioctl replace -f samples/apps/bookinfo/route-rule-reviews
   -50-v3.yaml
   ```

   注意这次使用的是 `replace` 命令，而不是 `create` ，因为该rule已经在前面创建过了。

2. 登出jason用户，或者删除测试规则，可以看到新的规则已经生效。

   删除测试规则。

   ```
   istioctl delete route-rule reviews-test-v2
   istioctl delete route-rule ratings-test-delay
   ```

   现在的规则就是刷新 `productpage` 页面，50%的概率看到红色星标的评论，50%的概率看不到星标。

   注意：因为使用Envoy sidecar的实现，你需要刷新页面很多次才能看到接近规则配置的概率分布，你可以将v3的概率修改为90%，这样刷新页面时，看到红色星标的概率更高。

3. 当v3版本的微服务稳定以后，就可以将100%的流量分摊到 `reviews:v3` 上了。

   ```
   istioctl replace -f samples/apps/bookinfo/route-rule-reviews
   -v3.yaml
   ```

   现在不论你使用什么用户登陆 `productpage` 页面，你都可以看到带红色星标评分的评论了。

安装和拓展 Istio mesh

前置条件

下面的操作说明需要您可以访问 kubernetes **1.7.3** 后更高版本 的集群，并且启用了 RBAC (基于角色的访问控制)。您需要安装了 **1.7.3** 或更高版本 的 `kubectl` 命令。如果您希望启用 自动注入 sidecar，您需要启用 kubernetes 集群的 alpha 功能。

> 注意：如果您安装了 Istio 0.1.x，在安装新版本前请先 卸载 它们（包括已启用 Istio 应用程序 Pod 中的 sidecar）。

- 取决于您的 kubernetes 提供商：
  - 本地安装 Istio，安装最新版本的 Minikube (version 0.22.1 或者更高)。
  - Google Container Engine
    - 使用 kubectl 获取证书 （使用您自己的集群的名字替换 `<cluster-name>` ，使用集群实际所在的位置替换 `<zone>` ）：

      ```
      gcloud container clusters get-credentials <cluster-name> --zone <zone> --project <project-name>
      ```

    - 将集群管理员权限授予当前用户（需要管理员权限才能为Istio创建必要的RBAC规则）：

      ```
      kubectl create clusterrolebinding cluster-admin-binding --clusterrole=cluster-admin --user=$(gcloud config get-value core/account)
      ```

  - IBM Bluemix Container Service
    - 使用 kubectl 获取证书 （使用您自己的集群的名字替换）：

      ```
      <cluster-name>
      ```

```
$(bx cs cluster-config <cluster-name>|grep "export KU
BECONFIG")
```

- Openshift Origin 3.7 或者以上版本：
  - 默认情况下，Openshift 不允许以 UID 0运行容器。为 Istio 的入口（ingress）和出口（egress）service account 启用使用UID 0运行的容器：

    ```
    oc adm policy add-scc-to-user anyuid -z istio-ingress
    -service-account -n istio-system
    oc adm policy add-scc-to-user anyuid -z istio-egress-
    service-account -n istio-system
    oc adm policy add-sc-to-user anyuid -z default -n ist
    io-system
    ```

  - 运行应用程序 Pod 的 service account 需要特权安全性上下文限制，以此作为 sidecar 注入的一部分:

    ```
    oc adm policy add-scc-to-user privileged -z default -
    n <target-namespace>
    ```

- 安装或升级 Kubernetes 命令行工具 kubectl 以匹配您的集群版本（1.7或以上版本）。

安装步骤

不论对于哪个 Istio 发行版，都安装到 `istio-system` namespace 下，即可以管理所有其它 namespace 下的微服务。

1. 到 Istio release 页面上，根据您的操作系统下载对应的发行版。如果您使用的是 MacOS 或者 Linux 系统，可以使用下面的额命令自动下载和解压最新的发行版：

   ```
   curl -L https://git.io/getLatestIstio | sh -
   ```

2. 解压安装文件，切换到文件所在目录。安装文件目录下包含：

- `install/` 目录下是 kubernetes 使用的 `.yaml` 安装文件
- `samples/` 目录下是示例程序
- `istioctl` 客户端二进制文件在 `bin` 目录下。 `istioctl` 文件用户手动注入 Envoy sidecar 代理、创建路由和策略等。
- `istio.VERSION` 配置文件

3. 切换到 istio 包的解压目录。例如 istio-0.2.7：

```
cd istio-0.2.7
```

4. 将 `istioctl` 客户端二进制文件加到 PATH 中。

   例如，在 MacOS 或 Linux 系统上执行下面的命令：

```
export PATH=$PWD/bin:$PATH
```

5. 安装 Istio 的核心部分。选择面两个 互斥 选项中的之一：

   a) 安装 Istio 的时候不启用 sidecar 之间的 TLS 双向认证：

   为具有现在应用程序的集群选择该选项，使用 Istio sidecar 的服务需要能够与非 Istio Kubernetes 服务以及使用 liveliness 和 readiness 探针、headless service 和 StatefulSet 的应用程序通信。

```
kubectl apply -f install/kubernetes/istio.yaml
```

   或者

   b) 安装 Istio 的时候启用 sidecar 之间的 TLS 双向认证：

```
kubectl apply -f install/kubernetes/istio-auth.yaml
```

   这两个选项都会创建 `istio-system` 命名空间以及所需的 RBAC 权限，并部署 Istio-Pilot、Istio-Mixer、Istio-Ingress、Istio-Egress 和 Istio-CA（证书颁发机构）。

6. 可选的：如果您的 kubernetes 集群开启了 alpha 功能，并想要启用 自动注入 sidecar，需要安装 Istio-Initializer：

```
kubectl apply -f install/kubernetes/istio-initializer.yaml
```

验证安装

1. 确认系列 kubernetes 服务已经部署了： `istio-pilot` 、 `istio-mixer` 、 `istio-ingress` 、 `istio-egress` ：

    ```
    kubectl get svc -n istio-system
    ```

    ```
    NAME             CLUSTER-IP      EXTERNAL-IP       PORT(S)
        AGE
    istio-egress     10.83.247.89    <none>            80/TCP
        5h
    istio-ingress    10.83.245.171   35.184.245.62     80:32730/T
    CP,443:30574/TCP    5h
    istio-pilot      10.83.251.173   <none>            8080/TCP,8
    081/TCP             5h
    istio-mixer      10.83.244.253   <none>            9091/TCP,9
    094/TCP,42422/TCP   5h
    ```

    注意：如果您运行的集群不支持外部负载均衡器（如 minikube）， `istio-ingress` 服务的 `EXTERNAL-IP` 显示 `<pending>` 。你必须改为使用 NodePort service 或者 端口转发方式来访问应用程序。

2. 确认对应的 Kubernetes pod 已部署并且所有的容器都启动并运行： `istio-pilot-*` 、 `istio-mixer-*` 、 `istio-ingress-*` 、 `istio-egress-*` 、 `istio-ca-*` ， `istio-initializer-*` 是可以选的。

    ```
    kubectl get pods -n istio-system
    ```

```
istio-ca-3657790228-j21b9              1/1       Running   0
       5h
istio-egress-1684034556-fhw89          1/1       Running   0
       5h
istio-ingress-1842462111-j3vcs         1/1       Running   0
       5h
istio-initializer-184129454-zdgf5      1/1       Running   0
       5h
istio-pilot-2275554717-93c43           1/1       Running   0
       5h
istio-mixer-2104784889-20rm8           2/2       Running   0
       5h
```

部署应用

您可以部署自己的应用或者示例应用程序如 BookInfo。 注意：应用程序必须使用 HTTP/1.1 或 HTTP/2.0 协议来传输 HTTP 流量，因为 HTTP/1.0 已经不再支持。

如果您启动了 Istio-Initializer，如上所示，您可以使用 `kubectl create` 直接部署应用。Istio-Initializer 会向应用程序的 pod 中自动注入 Envoy 容器：

```
kubectl create -f <your-app-spec>.yaml
```

如果您没有安装 Istio-initializer 的话，您必须使用 istioctl kube-inject 命令在部署应用之前向应用程序的 pod 中手动注入 Envoy 容器：

```
kubectl create -f <(istioctl kube-inject -f <your-app-spec>.yaml)
```

卸载

- 卸载 Istio initializer:

  如果您安装 Isto 的时候启用了 initializer，请卸载它：

  ```
  kubectl delete -f install/kubernetes/istio-initializer.yaml
  ```

- 卸载 Istio 核心组件。对于某一 Istio 版本，删除 RBAC 权限， `istio-system` namespace，和该命名空间的下的各层级资源。

  不必理会在层级删除过程中的各种报错，因为这些资源可能已经被删除的。

  a) 如果您在安装 Istio 的时候关闭了 TLS 双向认证：

  ```
  kubectl delete -f install/kubernetes/istio.yaml
  ```

  或者

  b) 如果您在安装 Istio 的时候启用到了 TLS 双向认证：

  ```
  kubectl delete -f install/kubernetes/istio-auth.yaml
  ```

安装 Istio Sidecar

Pod Spec 需满足的条件

为了成为 Service Mesh 中的一部分，kubernetes 集群中的每个 Pod 都必须满足如下条件：

1. **Service** 注解：每个 pod 都必须只属于某一个 Kubernetes Service （当前不支持一个 pod 同时属于多个 service）。
2. 命名的端口：Service 的端口必须命名。端口的名字必须遵循如下格式 `<protocol>[-<suffix>]` ，可以是http、http2、 grpc、 mongo、 或者 redis 作为 `<protocol>` ，这样才能使用 Istio 的路由功能。例如 `name: http2-foo` 和 `name: http` 都是有效的端口名称，而 `name: http2foo` 不是。如果端口的名称是不可识别的前缀或者未命名，那么该端口上的流量就会作为普通的 TCP 流量（除非使用 `Protocol: UDP` 明确声明使用 UDP 端口）。
3. 带有 **app label** 的 **Deployment**：我们建议 kubernetes 的 `Deploymenet` 资源的配置文件中为 Pod 明确指定 `app` label。每个Deployment 的配置中都需要有个不同的有意义的 `app` 标签。 `app` label 用于在分布式坚重中添加上下文信息。
4. **Mesh** 中的每个 **pod** 里都有一个 **Sidecar**： 最后，Mesh 中的每个 pod 都必须

运行与 Istio 兼容的sidecar。下面部分介绍了将 sidecar 注入到 pod 中的两种方法：使用 `istioctl` 命令行工具手动注入，或者使用 istio initializer 自动注入。注意 sidecar 不涉及到容器间的流量，因为它们都在同一个 pod 中。

手动注入 sidecar

`istioctl` 命令行中有一个称为 kube-inject 的便利工具，使用它可以将 Istio 的 sidecar 规范添加到 kubernetes 工作负载的规范配置中。与 Initializer 程序不同，`kube-inject` 只是将 YAML 规范转换成包含 Istio sidecar 的规范。您需要使用标准的工具如 `kubectl` 来部署修改后的 YAML。例如，以下命令将 sidecar 添加到 sleep.yaml 文件中指定的 pod 中，并将修改后的规范提交给 kubernetes：

```
kubectl apply -f <(istioctl kube-inject -f samples/sleep/sleep.y
aml)
```

示例

我们来试一试将 Istio sidecar 注入到 sleep 服务中去。

```
kubectl apply -f <(istioctl kube-inject -f samples/sleep/sleep.y
aml)
```

Kube-inject 子命令将 Istio sidecar 和 init 容器注入到 deployment 配置中，转换后的输出如下所示：

```
... 略过 ...
---
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  annotations:
    sidecar.istio.io/status: injected-version-root@69916ebba0fc-
0.2.6-081ffece00c82cb9de33cd5617682999aee5298d
  name: sleep
spec:
  replicas: 1
  template:
    metadata:
      annotations:
        sidecar.istio.io/status: injected-version-root@69916ebba
0fc-0.2.6-081ffece00c82cb9de33cd5617682999aee5298d
      labels:
        app: sleep
    spec:
      containers:
      - name: sleep
        image: tutum/curl
        command: ["/bin/sleep","infinity"]
        imagePullPolicy: IfNotPresent
      - name: istio-proxy
        image: docker.io/istio/proxy_debug:0.2.6
        args:
        ... 略过 ...
      initContainers:
      - name: istio-init
        image: docker.io/istio/proxy_init:0.2.6
        imagePullPolicy: IfNotPresent
        args:
        ... 略过 ...
---
```

注入 sidecar 的关键在于 `initContainers` 和 istio-proxy 容器。为了简洁起见，上述输出有所省略。

验证 sleep deployment 中包含 sidecar。injected-version 对应于注入的 sidecar 镜像的版本和镜像的 TAG。在您的设置的可能会有所不同。

```
echo $(kubectl get deployment sleep -o jsonpath='{.metadata.anno
tations.sidecar\.istio\.io\/status}')
```

```
injected-version-9c7c291eab0a522f8033decd0f5b031f5ed0e126
```

你可以查看包含注入的容器和挂载的 volume 的完整 deployment 信息。

```
kubectl get deployment sleep -o yaml
```

自动注入 sidecar

Istio sidecar 可以在部署之前使用 Kubernetes 中一个名为 Initializer 的 Alpha 功能自动注入到 Pod 中。

> 注意：Kubernetes InitializerConfiguration没有命名空间，适用于整个集群的工作负载。不要在共享测试环境中启用此功能。

前置条件

Initializer 需要在集群设置期间显示启用，如 此处 所述。 假设集群中已启用 RBAC，则可以在不同环境中启用初始化程序，如下所示：

- *GKE*

  ```
  gcloud container clusters create NAME \
      --enable-kubernetes-alpha \
      --machine-type=n1-standard-2 \
      --num-nodes=4 \
      --no-enable-legacy-authorization \
      --zone=ZONE
  ```

- *IBM Bluemix* kubernetes v1.7.4 或更高版本的集群已默认启用 initializer。

- *Minikube*

  Minikube v0.22.1 或更高版本需要为 GenericAdmissionWebhook 功能配置适当的证书。获取最新版本： https://github.com/kubernetes/minikube/releases.

  ```
  minikube start \
      --extra-config=apiserver.Admission.PluginNames="Initializers,NamespaceLifecycle,LimitRanger,ServiceAccount,DefaultStorageClass,GenericAdmissionWebhook,ResourceQuota" \
      --kubernetes-version=v1.7.5
  ```

安装

您现在可以从 Istio 安装根目录设置 Istio Initializer。

```
kubectl apply -f install/kubernetes/istio-initializer.yaml
```

将会创建下列资源：

1. `istio-sidecar` InitializerConfiguration 资源，指定 Istio sidecar 注入的资源。默认情况下 Istio sidecar 将被注入到 `deployment` 、 `statefulset` 、 `job` 和 `daemonset` 中。
2. `istio-inject` ConfigMap，initializer 的默认注入策略，一组初始化 namespace，以及注入时使用的模版参数。这些配置的详细说明请参考 配置选项。
3. `istio-initializer` Deployment，运行 initializer 控制器。
4. `istio-initializer-service-account` ServiceAccount，用于 `istio-initializer` deployment。 `ClusterRole` 和 `ClusterRoleBinding` 在 `install/kubernetes/istio.yaml` 中定义。注意所有的资源类型都需要有 `initialize` 和 `patch` 。正式处于这个原因，initializer 要作为 deployment 的一部分来运行而不是嵌入到其它控制器中，例如 istio-pilot。

验证

为了验证 sidecar 是否成功注入，为上面的 sleep 服务创建 deployment 和 service。

```
kubectl apply -f samples/sleep/sleep.yaml
```

验证 sleep deployment 中包含 sidecar。injected-version 对应于注入的 sidecar 镜像的版本和镜像的 TAG。在您的设置的可能会有所不同。

```
$ echo $(kubectl get deployment sleep -o jsonpath='{.metadata.annotations.sidecar\.istio\.io\/status}')
```

```
injected-version-9c7c291eab0a522f8033decd0f5b031f5ed0e126
```

你可以查看包含注入的容器和挂载的 volume 的完整 deployment 信息。

```
kubectl get deployment sleep -o yaml
```

了解发生了什么

以下是将工作负载提交给 Kubernetes 后发生的情况：

1) kubernetes 将 `sidecar.initializer.istio.io` 添加到工作负载的 pending initializer 列表中。

2) istio-initializer 控制器观察到有一个新的未初始化的工作负载被创建了。pending initializer 列表中的第一个个将作为 `sidecar.initializer.istio.io` 的名称。

3) istio-initializer 检查它是否负责初始化 namespace 中的工作负载。如果没有为该 namespace 配置 initializer，则不需要做进一步的工作，而且 initializer 会忽略工作负载。默认情况下，initializer 负责所有的 namespace（参考 配置选项）。

4) istio-initializer 将自己从 pending initializer 中移除。如果 pending initializer 列表非空，则 Kubernetes 不回结束工作负载的创建。错误配置的 initializer 意味着破损的集群。

5) istio-initializer 检查 mesh 的默认注入策略，并检查所有单个工作负载的策略负载值，以确定是否需要注入 sidecar。

6) istio-initializer 向工作负载中注入 sidecar 模板，然后通过 PATCH 向 kubernetes 提交。

7) kubernetes 正常的完成了工作负载的创建，并且工作负载中已经包含了注入的 sidecar。

配置选项

istio-initializer 具有用于注入的全局默认策略以及每个工作负载覆盖配置。全局策略由 `istio-inject` ConfigMap 配置（请参见下面的示例）。Initializer pod 必须重新启动以采用新的配置更改。

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: istio-inject
  namespace: istio-system
data:
  config: |-
    policy: "enabled"
    namespaces: [""] # everything, aka v1.NamepsaceAll, aka clus
ter-wide
    # excludeNamespaces: ["ns1", "ns2"]
    initializerName: "sidecar.initializer.istio.io"
    params:
      initImage: docker.io/istio/proxy_init:0.2.6
      proxyImage: docker.io/istio/proxy:0.2.6
      verbosity: 2
      version: 0.2.6
      meshConfigMapName: istio
      imagePullPolicy: IfNotPresent
```

下面是配置中的关键参数：

1. **policy**

   `off` - 禁用 initializer 修改资源。pending 的 `sidecar.initializer.istio.io` initializer 将被删除以避免创建阻塞资源。

   `disable` - initializer 不会注入 sidecar 到 watch 的所有 namespace 的资源中。启用 sidecar 注入请将 `sidecar.istio.io/inject` 注解的值设置为 `true`。

   `enable` - initializer 将会注入 sidecar 到 watch 的所有 namespace 的资源中。禁用 sidecar 注入请将 `sidecar.istio.io/inject` 注解的值设置为 `false`。

2. **namespaces**

   要 watch 和初始化的 namespace 列表。特殊的 `""` namespace 对应于 `v1.NamespaceAll` 并配置初始化程序以初始化所有 namespace。`kube-system`、`kube-publice` 和 `istio-system` 被免除初始化。

1. **excludeNamespaces**

从 Istio initializer 中排除的 namespace 列表。不可以定义为 `v1.NamespaceAll` 或者与 `namespaces` 一起定义。

1. **initializerName**

   这必须与 InitializerConfiguration 中 initializer 设定项的名称相匹配。Initializer 只处理匹配其配置名称的工作负载。

1. **params**

   这些参数允许您对注入的 sidecar 进行有限的更改。更改这些值不会影响已部署的工作负载。

重写自动注入

单个工作负载可以通过使用 `sidecar.istio.io/inject` 注解重写全局策略。如果注解被省略，则使用全局策略。

如果注解的值是 `true` ，则不管全局策略如何，sidecar 都将被注入。

如果注解的值是 `false` ，则不管全局策略如何，sidecar 都不会被注入。

下表显示全局策略和每个工作负载覆盖的组合。

policy	workload annotation	injected
off	N/A	no
disabled	omitted	no
disabled	false	no
disabled	true	yes
enabled	omitted	yes
enabled	false	no
enabled	true	yes

例如，即使全局策略是 `disable` ，下面的 deployment 也会被注入sidecar。

```yaml
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: myapp
  annotations:
    sidecar.istio.io/inject: "true"
spec:
  replicas: 1
  template:
    ...
```

这是在包含 Istio 和非 Istio 服务的混合群集中使用自动注入的好方法。

卸载 Initializer

运行下面的命令，删除 Istio initializer：

```
kubectl delete -f install/kubernetes/istio-initializer.yaml
```

注意上述命令并不会删除已注入到 Pod 中的 sidecar。要想删除这些 sidecar，需要在不使用 initializer 的情况下重新部署这些 pod。

拓展 Istio Mesh

将虚拟机或裸机集成到部署在 kubernetes 集群上的 Istio mesh 中的说明。

前置条件

- 按照 安装指南 在 kubernetes 集群上安装 Istio service mesh。
- 机器必须具有到 mesh 端点的 IP 地址连接。这通常需要一个 VPC 或者 VPN，以及一个向端点提供直接路由（没有 NAT 或者防火墙拒绝）的容器网络。及其不需要访问有 Kubernetes 分配的 cluster IP。
- 虚拟机必须可以访问到 Istio 控制平面服务（如Pilot、Mixer、CA）和 Kubernetes DNS 服务器。通常使用 内部负载均衡器 来实现。

  您也可以使用 NodePort，在虚拟机上运行 Istio 的组件，或者使用自定义网络配置，有几个单独的文档会涵盖这些高级配置。

安装步骤

安装过程包括准备用于拓展的 mesh 和安装和配置虚拟机。

install/tools/setupMeshEx.sh ：这是一个帮助大家设置 kubernetes 环境的示例脚本。检查脚本内容和支持的环境变量（如 GCP_OPTS）。

install/tools/setupIstioVM.sh：这是一个用于配置主机环境的示例脚本。 您应该根据您的配置工具和DNS要求对其进行自定义。

准备要拓展的 Kubernetes 集群：

- 为 Kube DNS、Pilot、Mixer 和 CA 安装内部负载均衡器（ILB）。每个云供应商的配置都有所不同，根据具体情况修改注解。

> 0.2.7 版本的 YAML 文件的 DNS ILB 的 namespace 配置不正确。 使用 这一个 替代。 `setupMeshEx.sh` 中也有错误。使用上面链接中的最新文件或者从 GitHub.com/istio/istio 克隆。

```
kubectl apply -f install/kubernetes/mesh-expansion.yaml
```

- 生成要部署到虚拟机上的 Istio `cluster.env` 配置。该文件中包含要拦截的 cluster IP 地址范围。

```
export GCP_OPTS="--zone MY_ZONE --project MY_PROJECT"
```

```
install/tools/setupMeshEx.sh generateClusterEnv MY_CLUSTER_NAME
```

该示例生成的文件：

```
cat cluster.env
```

```
ISTIO_SERVICE_CIDR=10.63.240.0/20
```

- 产生虚拟机使用的 DNS 配置文件。这样可以让虚拟机上的应用程序解析到集群中的服务名称，这些名称将被 sidecar 拦截和转发。

```
# Make sure your kubectl context is set to your cluster
install/tools/setupMeshEx.sh generateDnsmasq
```

该示例生成的文件：

```
cat kubedns


server=/svc.cluster.local/10.150.0.7
address=/istio-mixer/10.150.0.8
address=/istio-pilot/10.150.0.6
address=/istio-ca/10.150.0.9
address=/istio-mixer.istio-system/10.150.0.8
address=/istio-pilot.istio-system/10.150.0.6
address=/istio-ca.istio-system/10.150.0.9
```

设置机器

例如，您可以使用下面的“一条龙”脚本复制和安装配置：

```
# 检查该脚本看看它是否满足您的需求
# 在 Mac 上，使用 brew install base64 或者 set BASE64_DECODE="/usr
/bin/base64 -D"
export GCP_OPTS="--zone MY_ZONE --project MY_PROJECT"


install/tools/setupMeshEx.sh machineSetup VM_NAME
```

或者等效得手动安装步骤如下：

------ 手动安装步骤开始 ------

- 将配置文件和 Istio 的 Debian 文件复制到要加入到集群的每台机器上。重命名为 `/etc/dnsmasq.d/kubedns` 和 `/var/lib/istio/envoy/cluster.env` 。
- 配置和验证 DNS 配置。需要安装 `dnsmasq` 或者直接将其添加到 `/etc/resolv.conf` 中，或者通过 DHCP 脚本。验证配置是否有效，检查虚拟机是否可以解析和连接到 pilot，例如：

在虚拟机或外部主机上：

```
host istio-pilot.istio-system
```

产生的消息示例：

```
# Verify you get the same address as shown as "EXTERNAL-IP" in '
kubectl get svc -n istio-system istio-pilot-ilb'
istio-pilot.istio-system has address 10.150.0.6
```

检查是否可以解析 cluster IP。实际地址取决您的 deployment：

```
host istio-pilot.istio-system.svc.cluster.local.
```

该示例产生的消息：

```
istio-pilot.istio-system.svc.cluster.local has address 10.63.247
.248
```

同样检查 istio-ingress：

```
host istio-ingress.istio-system.svc.cluster.local.
```

该示例产生的消息：

```
istio-ingress.istio-system.svc.cluster.local has address 10.63.2
43.30
```

- 验证连接性，检查迅即是否可以连接到 Pilot 的端点：

```
curl 'http://istio-pilot.istio-system:8080/v1/registration/istio
-pilot.istio-system.svc.cluster.local|http-discovery'
```

```
{
  "hosts": [
   {
    "ip_address": "10.60.1.4",
    "port": 8080
   }
  ]
}
```

```
# 在虚拟机上使用上面的地址。将直接连接到运行 istio-pilot 的 pod。
curl 'http://10.60.1.4:8080/v1/registration/istio-pilot.istio-system.svc.cluster.local|http-discovery'
```

- 提取出实话 Istio 认证的 secret 并将它复制到机器上。Istio 的默认安装中包括 CA，即使是禁用了自动 `mTLS` 设置（她为每个 service account 创建 secret，secret 命名为 `istio.<serviceaccount>`）也会生成 Istio secret。建议您执行此步骤，以便日后启用 mTLS，并升级到默认启用 mTLS 的未来版本。

```
# ACCOUNT 默认是 'default'，SERVICE_ACCOUNT 是环境变量
# NAMESPACE 默认为当前 namespace，SERVICE_NAMESPACE 是环境变量
# （这一步由 machineSetup 完成）
# 在 Mac 上执行 brew install base64 或者 set BASE64_DECODE="/usr/bin/base64 -D"
install/tools/setupMeshEx.sh machineCerts ACCOUNT NAMESPACE
```

生成的文件（`key.pem`，`root-cert.pem`，`cert-chain.pem`）必须拷贝到每台主机的 /etc/certs 目录，并且让 istio-proxy 可读。

- 安装 Istio Debian 文件，启动 `istio` 和 `istio-auth-node-agent` 服务。从 github releases 获取 Debian 安装包：

```
# 注意：在软件源配置好后，下面的额命令可以使用 'apt-get' 命令替代。

source istio.VERSION # defines version and URLs env var
curl -L ${PILOT_DEBIAN_URL}/istio-agent.deb > ${ISTIO_STAGIN
G}/istio-agent.deb
curl -L ${AUTH_DEBIAN_URL}/istio-auth-node-agent.deb > ${IST
IO_STAGING}/istio-auth-node-agent.deb
curl -L ${PROXY_DEBIAN_URL}/istio-proxy.deb > ${ISTIO_STAGIN
G}/istio-proxy.deb

dpkg -i istio-proxy-envoy.deb
dpkg -i istio-agent.deb
dpkg -i istio-auth-node-agent.deb

systemctl start istio
systemctl start istio-auth-node-agent
```

------ 手动安装步骤结束 ------

安装完成后，机器就能访问运行在 Kubernetes 集群上的服务或者其他的 mesh 拓展的机器。

```
# 假设您在 'bookinfo' namespace 下安装的 bookinfo
curl productpage.bookinfo.svc.cluster.local:9080
```

```
... html content ...
```

检查进程是否正在运行：

```
ps aux |grep istio
```

```
root      6941  0.0  0.2  75392 16820 ?        Ssl  21:32   0:00
 /usr/local/istio/bin/node_agent --logtostderr
root      6955  0.0  0.0  49344  3048 ?        Ss   21:32   0:00
su -s /bin/bash -c INSTANCE_IP=10.150.0.5 POD_NAME=demo-vm-1 POD
_NAMESPACE=default exec /usr/local/bin/pilot-agent proxy > /var/
log/istio/istio.log istio-proxy
istio-p+  7016  0.0  0.1 215172 12096 ?        Ssl  21:32   0:00
 /usr/local/bin/pilot-agent proxy
istio-p+  7094  4.0  0.3  69540 24800 ?        Sl   21:32   0:37
 /usr/local/bin/envoy -c /etc/istio/proxy/envoy-rev1.json --rest
art-epoch 1 --drain-time-s 2 --parent-shutdown-time-s 3 --servic
e-cluster istio-proxy --service-node sidecar~10.150.0.5~demo-vm-
1.default~default.svc.cluster.local
```

检查 Istio auth-node-agent 是否健康：

```
sudo systemctl status istio-auth-node-agent
```

```
● istio-auth-node-agent.service - istio-auth-node-agent: The Ist
io auth node agent
   Loaded: loaded (/lib/systemd/system/istio-auth-node-agent.ser
vice; disabled; vendor preset: enabled)
   Active: active (running) since Fri 2017-10-13 21:32:29 UTC; 9
s ago
     Docs: http://istio.io/
 Main PID: 6941 (node_agent)
    Tasks: 5
   Memory: 5.9M
      CPU: 92ms
   CGroup: /system.slice/istio-auth-node-agent.service
           └─6941 /usr/local/istio/bin/node_agent --logtostderr

Oct 13 21:32:29 demo-vm-1 systemd[1]: Started istio-auth-node-ag
ent: The Istio auth node agent.
Oct 13 21:32:29 demo-vm-1 node_agent[6941]: I1013 21:32:29.46931
4    6941 main.go:66] Starting Node Agent
Oct 13 21:32:29 demo-vm-1 node_agent[6941]: I1013 21:32:29.46936
5    6941 nodeagent.go:96] Node Agent starts successfully.
Oct 13 21:32:29 demo-vm-1 node_agent[6941]: I1013 21:32:29.48332
4    6941 nodeagent.go:112] Sending CSR (retrial #0) ...
Oct 13 21:32:29 demo-vm-1 node_agent[6941]: I1013 21:32:29.86257
5    6941 nodeagent.go:128] CSR is approved successfully. Will r
enew cert in 29m59.137732603s
```

在拓展的 mesh 中的机器上运行服务

- 配置 sidecar 拦截端口。在 `/var/lib/istio/envoy/sidecar.env` 中通过 `ISTIO_INBOUND_PORTS` 环境变量配置。

  例如（运行服务的虚拟机）：

  ```
  echo "ISTIO_INBOUND_PORTS=27017,3306,8080" > /var/lib/istio
  /envoy/sidecar.env
  systemctl restart istio
  ```

- 手动配置 selector-less 的 service 和 endpoint。“selector-less” service 用于那些不依托 Kubernetes pod 的 service。

  例如，在有权限的机器上修改 Kubernetes 中的 service：

  ```
  # istioctl register servicename machine-ip portname:port
  istioctl -n onprem register mysql 1.2.3.4 3306
  istioctl -n onprem register svc1 1.2.3.4 http:7000
  ```

安装完成后，Kubernetes pod 和其它 mesh 扩展将能够访问集群上运行的服务。

整合到一起

请参阅 拓展 BookInfo Mesh 指南。

部署 bookinfo 示例应用

该示例部署由四个单独的微服务组成的简单应用程序，用于演示Istio服务网格的各种功能。

概况

在本示例中，我们将部署一个简单的应用程序，显示书籍的信息，类似于网上书店的书籍条目。在页面上有书籍的描述、详细信息（ISBN、页数等）和书评。

BookInfo 应用程序包括四个独立的微服务：

- productpage：productpage(产品页面)微服务，调用 *details* 和 *reviews* 微服务

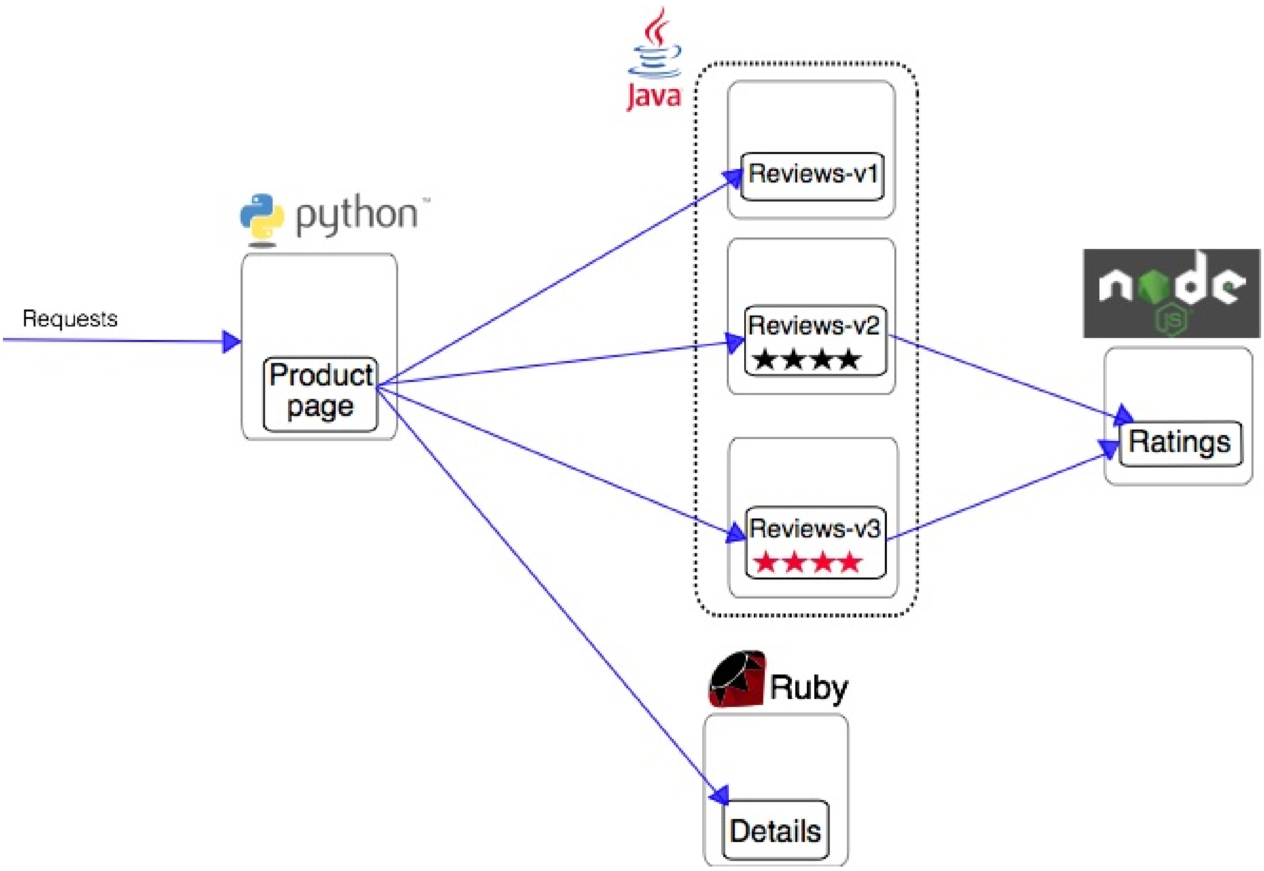
来填充页面。

- details：details 微服务包含书籍的详细信息。
- reviews：reviews 微服务包含书籍的点评。它也调用 *ratings* 微服务。
- ratings：ratings 微服务包含随书评一起出现的评分信息。

有3个版本的 reviews 微服务：

- 版本v1不调用 ratings 服务。
- 版本v2调用 ratings ，并将每个评级显示为1到5个黑色星。
- 版本v3调用 ratings ，并将每个评级显示为1到5个红色星。

应用程序的端到端架构如下所示。



图片 - *BookInfo*

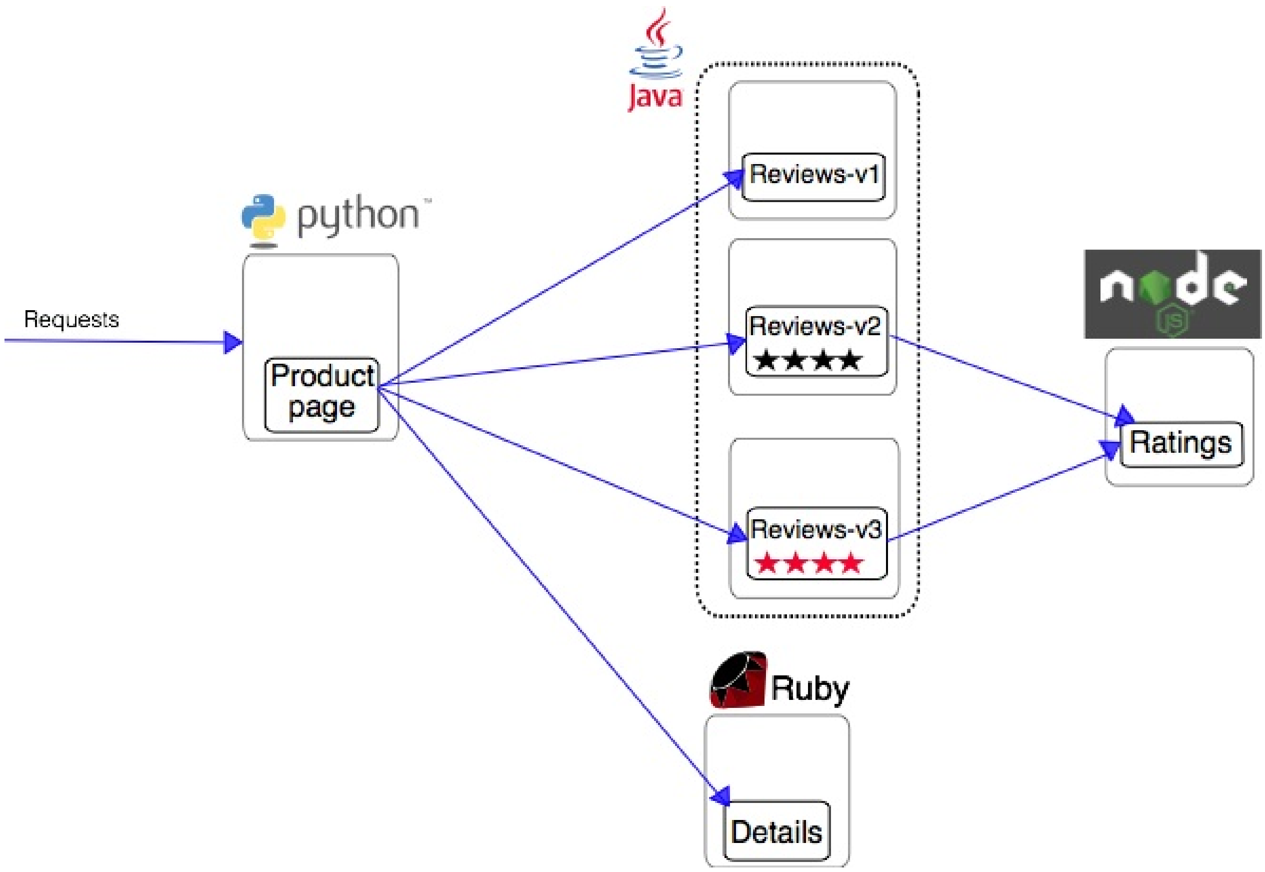
该应用程序是多语言构建的，即这些微服务是用不同的语言编写的。值得注意的是，这些服务与 Istio 没有任何依赖关系，单这是个有趣的 Service Mesh 示例，特别是因为评论服务和众多的语言和版本。

开始之前

如果您还没有这样做，请按照与您的平台 安装指南 对应的说明安装Istio。

部署应用程序

使用 Istio 运行应用程序示例不需要修改应用程序本身。相反，我们只需要在支持 Istio 的环境中配置和运行服务， Envoy sidecar 将会注入到每个服务中。所需的命令和配置根据运行时环境的不同而有所不同，但在所有情况下，生成的部署将如下所示：



图片 - *BookInfo*

所有的微服务都将与一个 Envoy sidecar 一起打包，拦截这些服务的入站和出站的调用请求，提供通过 Istio 控制平面从外部控制整个应用的路由，遥测收集和策略执行所需的 hook。

要启动该应用程序，请按照以下对应于您的 Istio 运行时环境的说明进行操作。

在 Kubernetes 中运行

> 注意：如果您使用 GKE，清确保您的集群至少有 4 个标准的 GKE 节点。如果您使用 Minikube，请确保您至少有 4GB 内存。

1. 将目录更改为 Istio 安装目录的根目录。

2. 构建应用程序容器：

   如果您使用 自动注入 **sidecar** 的方式部署的集群，那么只需要使用 `kubectl` 命令部署服务：

   ```
   kubectl apply -f samples/bookinfo/kube/bookinfo.yaml
   ```

   如果您使用 手动注入 **sidecar** 的方式部署的集群，清使用下面的命令：

   ```
   kubectl apply -f <(istioctl kube-inject -f samples/apps/book
   info/bookinfo.yaml)
   ```

   请注意，该 `istioctl kube-inject` 命令用于在创建部署之前修改 `bookinfo.yaml` 文件。这将把 Envoy 注入到 Kubernetes 资源。

   上述命令启动四个微服务并创建网关入口资源，如下图所示。3 个版本的评论的服务 v1、v2、v3 都已启动。

   > 请注意在实际部署中，随着时间的推移部署新版本的微服务，而不是同时部署所有版本。

3. 确认所有服务和 pod 已正确定义并运行：

   ```
   kubectl get services
   ```

   这将产生以下输出：

```
NAME                           CLUSTER-IP    EXTERNAL-IP    PORT(S)
              AGE
details                        10.0.0.31     <none>         9080/T
CP              6m
istio-ingress                  10.0.0.122    <pending>      80:315
65/TCP          8m
istio-pilot                    10.0.0.189    <none>         8080/T
CP              8m
istio-mixer                    10.0.0.132    <none>         9091/T
CP,42422/TCP    8m
kubernetes                     10.0.0.1      <none>         443/TC
P               14d
productpage                    10.0.0.120    <none>         9080/T
CP              6m
ratings                        10.0.0.15     <none>         9080/T
CP              6m
reviews                        10.0.0.170    <none>         9080/T
CP              6m
```

而且

```
kubectl get pods
```

将产生:

```
NAME                                        READY     STATUS
    RESTARTS   AGE
details-v1-1520924117-48z17                 2/2       Runnin
g    0          6m
istio-ingress-3181829929-xrrk5              1/1       Runnin
g    0          8m
istio-pilot-175173354-d6jm7                 2/2       Runnin
g    0          8m
istio-mixer-3883863574-jt09j                2/2       Runnin
g    0          8m
productpage-v1-560495357-jk1lz              2/2       Runnin
g    0          6m
ratings-v1-734492171-rnr5l                  2/2       Runnin
g    0          6m
reviews-v1-874083890-f0qf0                  2/2       Runnin
g    0          6m
reviews-v2-1343845940-b34q5                 2/2       Runnin
g    0          6m
reviews-v3-1813607990-8ch52                 2/2       Runnin
g    0          6m
```

确定 ingress IP 和端口

1. 如果您的 kubernetes 集群环境支持外部负载均衡器的话，可以使用下面的命令获取 ingress 的IP地址：

   ```
   kubectl get ingress -o wide
   ```

   输出如下所示：

   ```
   NAME       HOSTS      ADDRESS                  PORTS      AGE
   gateway    *          130.211.10.121           80         1d
   ```

   Ingress 服务的地址是：

   ```
   export GATEWAY_URL=130.211.10.121:80
   ```

2. GKE：如果服务无法获取外部 IP， `kubectl get ingress -o wide` 会显示工作节点的列表。在这种情况下，您可以使用任何地址以及 NodePort 访问入口。但是，如果集群具有防火墙，则还需要创建防火墙规则以允许TCP流量到NodePort，您可以使用以下命令创建防火墙规则：

   ```
   export GATEWAY_URL=<workerNodeAddress>:$(kubectl get svc ist
   io-ingress -n istio-system -o jsonpath='{.spec.ports[0].node
   Port}')
   gcloud compute firewall-rules create allow-book --allow tcp:
   $(kubectl get svc istio-ingress -n istio-system -o jsonpath=
   '{.spec.ports[0].nodePort}')
   ```

3. IBM Bluemix Free Tier：在免费版的 Bluemix 的 kubernetes 集群中不支持外部负载均衡器。您可以使用工作节点的公共 IP，并通过 NodePort 来访问 ingress。工作节点的公共 IP可以通过如下命令获取：

   ```
   bx cs workers <cluster-name or id>
   export GATEWAY_URL=<public IP of the worker node>:$(kubectl
   get svc istio-ingress -n istio-system -o jsonpath='{.spec.po
   rts[0].nodePort}')
   ```

4. Minikube：Minikube 不支持外部负载均衡器。您可以使用 ingress 服务的主机 IP 和 NodePort 来访问 ingress：

```
export GATEWAY_URL=$(kubectl get po -l istio=ingress -o 'jsonpath={.items[0].status.hostIP}'):$(kubectl get svc istio-ingress -o 'jsonpath={.spec.ports[0].nodePort}')
```

在 Consul 或 Eureka 环境下使用 Docker 运行

1. 切换到 Istio 的安装根目录下。
2. 启动应用程序容器。
   i. 执行下面的命令测试 Consul：

   ```
   docker-compose -f samples/bookinfo/consul/bookinfo.yaml up -d
   ```

   ii. 执行下面的命令测试 Eureka：

   ```
   docker-compose -f samples/bookinfo/eureka/bookinfo.yaml up -d
   ```

3. 确认所有容器都在运行：

   ```
   docker ps -a
   ```

   > 如果 Istio Pilot 容器终止了，重新执行上面的命令重新运行。

4. 设置 `GATEWAY_URL`：

   ```
   export GATEWAY_URL=localhost:9081
   ```

下一步

使用以下 `curl` 命令确认 BookInfo 应用程序正在运行:

```
curl -o /dev/null -s -w "%{http_code}\n" http://${GATEWAY_URL}/p
roductpage
```

```
200
```

你也可以通过在浏览器中打开 `http://$GATEWAY_URL/productpage` 页面访问 Bookinfo 网页。如果您多次刷新浏览器将在 productpage 中看到评论的不同的版本，它们会按照 round robin（红星、黑星、没有星星）的方式展现，因为我们还没有使用 Istio 来控制版本的路由。

现在，您可以使用此示例来尝试 Istio 的流量路由、故障注入、速率限制等功能。要继续的话，请参阅 Istio 指南，具体取决于您的兴趣。智能路由 是初学者入门的好方式。

清理

在完成 BookInfo 示例后，您可以卸载它，如下所示：

卸载 **Kubernetes** 环境

1. 删除路由规则，终止应用程序 pod

    ```
    samples/bookinfo/kube/cleanup.sh
    ```

2. 确认关闭

    ```
    istioctl get routerules   #-- there should be no more routin
    g rules
    kubectl get pods          #-- the BookInfo pods should be de
    leted
    ```

卸载 **docker** 环境

1. 删除路由规则和应用程序容器

    i. 若使用 Consul 环境安装，执行下面的命令：

```
samples/bookinfo/consul/cleanup.sh
```

ii. 若使用 Eureka 环境安装，执行下面的命令：

```
samples/bookinfo/eureka/cleanup.sh
```

2. 确认清理完成：

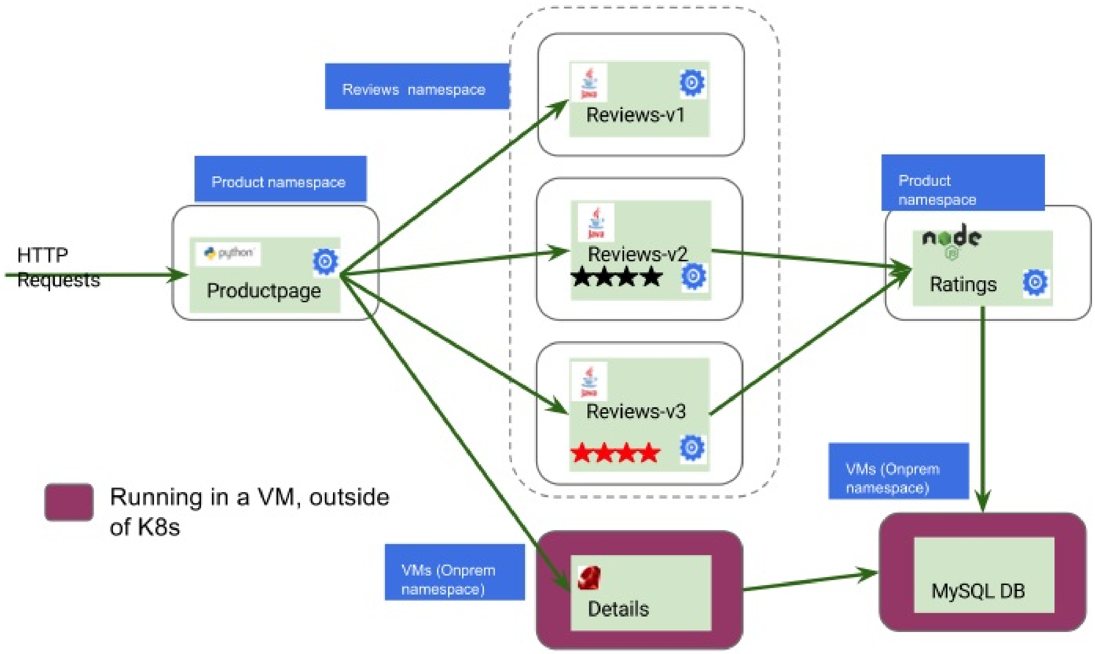
```
istioctl get routerules   #-- there should be no more routin
g rules
docker ps -a              #-- the BookInfo containers should
 be delete
```

集成虚拟机

该示例跨越 Kubernetes 集群和一组虚拟机上部署 Bookinfo 服务，描述如何使用 Istio service mesh 将此基础架构以单一 mesh 的方式操控。

> 注意：本文档还在建设中，并且只在 Google Cloud Platform 上进行过测试。在 IBM Bluemix 或其它平台上，pod 的 overlay 网络跟虚拟机的网络是隔离的。 即使使用 Istio，虚拟机也不能直接与 Kubernetes Pod 进行通信。

概览



图片 - *Bookinfo*应用的拓展*Mesh*

开始之前

- 按照 安装指南 上的步骤部署 Istio。
- 部署 BookInfo 示例应用程序（在 `bookinfo` namespace 下）。

- 在 Istio 集群相同的项目下创建名为 `vm-1` 的虚拟机，并 加入到 Mesh。

在虚拟机上运行 mysql

我们将首先在虚拟机上安装 mysql，将其配置成评分服务的后台存储。

在虚拟机上执行：

```
sudo apt-get update && sudo apt-get install -y mariadb-server
sudo mysql
# 授权 root 用户访问
GRANT ALL PRIVILEGES ON *.* TO 'root'@'localhost' IDENTIFIED BY
'password' WITH GRANT OPTION;
quit;
sudo systemctl restart mysql
```

关于 Mysql 的详细配置请见： Mysql。

在虚拟机上执行下面的命令，向 mysql 中添加评分数据库。

```
# 向 mysql 数据库中添加评分数据库
curl -q https://raw.githubusercontent.com/istio/istio/master/sam
ples/bookinfo/src/mysql/mysqldb-init.sql | mysql -u root -ppassw
ord
```

为了便于直观地查看bookinfo应用程序输出的差异，可以更改使用以下命令生成的评分：

```
# 查看评分
mysql -u root -ppassword test -e "select * from ratings;"
+----------+--------+
| ReviewID | Rating |
+----------+--------+
|        1 |      5 |
|        2 |      4 |
+----------+--------+
# 修改评分
mysql -u root -ppassword test -e  "update ratings set rating=1
where reviewid=1;select * from ratings;"
+----------+--------+
| ReviewID | Rating |
+----------+--------+
|        1 |      1 |
|        2 |      4 |
+----------+--------+
```

找出将添加到 mesh 中的虚拟机的 IP 地址

在虚拟机上执行：

```
hostname -I
```

将 mysql 服务注册到 mesh 中

在一台可以访问 `istioctl` 命令的主机上，注册该虚拟机和 mysql db service：

```
istioctl register -n vm mysqldb <ip-address-of-vm> 3306
# 示例输出
$ istioctl register mysqldb 192.168.56.112 3306
I1015 22:24:33.846492   15465 register.go:44] Registering for se
rvice 'mysqldb' ip '192.168.56.112', ports list [{3306 mysql}]
I1015 22:24:33.846550   15465 register.go:49] 0 labels ([]) and
1 annotations ([alpha.istio.io/kubernetes-serviceaccounts=default
])
W1015 22:24:33.866410   15465 register.go:123] Got 'services "my
sqldb" not found' looking up svc 'mysqldb' in namespace 'default'
, attempting to create it
W1015 22:24:33.904162   15465 register.go:139] Got 'endpoints "m
ysqldb" not found' looking up endpoints for 'mysqldb' in namespa
ce 'default', attempting to create them
I1015 22:24:33.910707   15465 register.go:180] No pre existing e
xact matching ports list found, created new subset {[{192.168.56
.112  <nil> nil}] [] [{mysql 3306 }]}
I1015 22:24:33.921195   15465 register.go:191] Successfully upda
ted mysqldb, now with 1 endpoints
```

集群管理

如果你之前在 kubernetes 上运行过 mysql，您需要将 kubernetes 的 mysql service 移除：

```
kubectl delete service mysql
```

执行 istioctl 来配置 service（在您的 admin 机器上）：

```
istioctl register mysql IP mysql:PORT
```

注意： `mysqldb` 虚拟机不需要也不应该有特别的 kubernetes 权限。

使用 mysql 服务

bookinfo 中的评分服务将使用机器上的数据库。要验证它是否有效，请创建使用虚拟机上的 mysql db 的评分服务的 v2 版本。然后指定强制评论服务使用评分 v2 版本的路由规则。

```
# 创建使用 mysql 后端的评分服务版本
istioctl kube-inject -n bookinfo -f samples/bookinfo/kube/bookin
fo-ratings-v2-mysql-vm.yaml | kubectl apply -n bookinfo -f -

# 强制 bookinfo 使用评分后端的路有规则
istioctl create -n bookinfo -f samples/bookinfo/kube/route-rule-
ratings-mysql-vm.yaml
```

您可以验证 bookinfo 应用程序的输出结果，显示来自 Reviewer1 的 1 星级和来自 Reviewer2 的 4 星级，或者更改虚拟机上的评分并查看结果。

同时，您还可以在 RawVM MySQL 的文档中找到一些故障排查和其它信息。

Istio 中 sidecar 的注入及示例

我们知道 Istio 通过向 Pod 中注入一个 sidecar 容器来将 Pod 纳入到 Istio service mesh 中的，那么这些 sidecar 容器的注入遵循什么样的规范，需要给每个 Pod 增加哪些配置信息才能纳入 Istio service mesh 中呢？这篇文章将给您答案。

Pod Spec 中需满足的条件

为了成为 Service Mesh 中的一部分，kubernetes 集群中的每个 Pod 都必须满足如下条件，这些规范不是由 Istio 自动注入的，而需要 生成 kubernetes 应用部署的 YAML 文件时需要遵守的：

1. **Service 关联**：每个 pod 都必须只属于某一个 Kubernetes Service （当前不支持一个 pod 同时属于多个 service）。
2. 命名的端口：Service 的端口必须命名。端口的名字必须遵循如下格式 `<protocol>[-<suffix>]` ，可以是 `http` 、 `http2` 、 `grpc` 、 `mongo` 、 或者 `redis` 作为 `<protocol>` ，这样才能使用 Istio 的路由功能。例如 `name: http2-foo` 和 `name: http` 都是有效的端口名称，而 `name: http2foo` 不是。如果端口的名称是不可识别的前缀或者未命名，那么该端口上的流量就会作为普通的 TCP 流量来处理（除非使用 `Protocol: UDP` 明确声明使用 UDP 端口）。
3. 带有 **app label 的 Deployment**：我们建议 kubernetes 的 `Deploymenet` 资源的配置文件中为 Pod 明确指定 `app` label。每个 Deployment 的配置中都需要有个与其他 Deployment 不同的含有意义的 `app` label。 `app` label 用于在分布式追踪中添加上下文信息。
4. **Mesh** 中的每个 **pod** 里都有一个 **Sidecar**：最后，Mesh 中的每个 pod 都必须运行与 Istio 兼容的 sidecar。以下部分介绍了将 sidecar 注入到 pod 中的两种方法：使用 `istioctl` 命令行工具手动注入，或者使用 Istio Initializer 自动注入。注意 sidecar 不涉及到流量，因为它们与容器位于同一个 pod 中。

将普通应用添加到 Istio service mesh 中

Istio官方的示例Bookinfo中并没有讲解如何将服务集成 Istio，只给出了 YAML 配置文件，而其中需要注意哪些地方都没有说明，假如我们自己部署的服务如何使用 Istio 呢？现在我们有如下两个普通应用（代码在 GitHub 上），它们都不具备微服务的高级特性，比如限流和熔断等，通过将它们部署到 kubernetes 并使用 Istio 来管理：

- k8s-app-monitor-test：用来暴露 json 格式的 metrics
- k8s-app-monitor-agent：访问上面那个应用暴露的 metrics 并生成监控图

这两个应用的 YAML 配置如下，其中包含了 Istio ingress 配置，并且符合 Istio 对 Pod 的 spec 配置所指定的规范。

k8s-app-monitor-istio-all-in-one.yaml文件

```yaml
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  annotations:
    kompose.cmd: kompose convert -f docker-compose.yaml
    kompose.version: 1.10.0 ()
  creationTimestamp: null
  labels:
    app: k8s-app-monitor-agent
  name: k8s-app-monitor-agent
spec:
  replicas: 1
  template:
    metadata:
      creationTimestamp: null
      labels:
        app: k8s-app-monitor-agent
    spec:
      containers:
      - env:
        - name: SERVICE_NAME
          value: k8s-app-monitor-test
        image: jimmysong/k8s-app-monitor-agent:749f547
        name: monitor-agent
        ports:
        - containerPort: 8888
      restartPolicy: Always
---
apiVersion: v1
kind: Service
metadata:
  annotations:
    kompose.cmd: kompose convert -f docker-compose.yaml
    kompose.version: 1.10.0 ()
```

```yaml
    creationTimestamp: null
    labels:
      app: k8s-app-monitor-agent
    name: k8s-app-monitor-agent
  spec:
    ports:
    - name: "http"
      port: 8888
      targetPort: 8888
    selector:
      app: k8s-app-monitor-agent
  ---
  apiVersion: extensions/v1beta1
  kind: Deployment
  metadata:
    annotations:
      kompose.cmd: kompose convert -f docker-compose.yaml
      kompose.version: 1.10.0 ()
    creationTimestamp: null
    labels:
      app: k8s-app-monitor-test
    name: k8s-app-monitor-test
  spec:
    replicas: 1
    template:
      metadata:
        creationTimestamp: null
        labels:
          app: k8s-app-monitor-test
      spec:
        containers:
        - image: jimmysong/k8s-app-monitor-test:9c935dd
          name: monitor-test
          ports:
          - containerPort: 3000
        restartPolicy: Always
  ---
  apiVersion: v1
  kind: Service
  metadata:
    annotations:
      kompose.cmd: kompose convert -f docker-compose.yaml
      kompose.version: 1.10.0 ()
    creationTimestamp: null
    labels:
      app: k8s-app-monitor-test
    name: k8s-app-monitor-test
  spec:
    ports:
    - name: "http"
      port: 3000
      targetPort: 3000
    selector:
```

```yaml
    app: k8s-app-monitor-test
---
## Istio ingress
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: k8s-app-monitor-agent-ingress
  annotations:
    kubernetes.io/ingress.class: "istio"
spec:
  rules:
  - http:
      paths:
      - path: /k8s-app-monitor-agent
        backend:
          serviceName: k8s-app-monitor-agent
          servicePort: 8888
```

其中有两点配置需要注意。

- `Deployment` 和 `Service` 中的 label 名字必须包含 `app`，zipkin 中的 tracing 需要使用到这个标签才能追踪
- `Service` 中的 `ports` 配置和必须包含一个名为 `http` 的 port，这样在 Istio ingress 中才能暴露该服务

注意：该 YAML 文件中 `annotations` 是因为我们一开始使用 `docker-compose` 部署在本地开发测试，后来再使用 kompose 将其转换为 kubernetes 可识别的 YAML 文件。

然后执行下面的命令就可以基于以上的 YAML 文件注入 sidecar 配置并部署到 kubernetes 集群中。

```
kubectl apply -n default -f <(istioctl kube-inject -f manifests/
istio/k8s-app-monitor-istio-all-in-one.yaml)
```

如何在本地启动 kubernetes 集群进行测试可以参考 kubernetes-vagrant-centos-cluster 中的说明。

Sidecar 注入说明

手动注入需要修改控制器的配置文件，如 deployment。通过修改 deployment 文件中的 pod 模板规范可实现该deployment 下创建的所有 pod 都注入 sidecar。添加/更新/删除 sidecar 需要修改整个 deployment。

自动注入会在 pod 创建的时候注入 sidecar，无需更改控制器资源。Sidecar 可通过以下方式更新：

- 选择性地手动删除 pod
- 系统得进行 deployment 滚动更新

手动或者自动注入都使用同样的模板配置。自动注入会从 `istio-system` 命名空间下获取 `istio-inject` 的 ConfigMap。手动注入可以通过本地文件或者 Configmap 。

参考

- Installing Istio Sidecar

如何参与 Istio 社区及注意事项

本文讲述了如何参与 Istio 社区和进行 Istio 开发时需要注意的事项。

工作组

绝大多数复杂的开源项目都是以工作组的方式组织的，要想为 Istio 社区做贡献可以加入到以下的工作组（Working Group）：

- API Management
- Config
- Environments
- Networking
- Performance & Scalability
- Policies & Telemetry
- Security
- Test & Release

代码规范

Istio 的代码规范沿用 CNCF 社区的代码规范。

开发指南

进行 Istio 开发之前需要做下面几件事情：

- 配置基础环境，如 Kubernetes
- 配置代码库、下载依赖和测试
- 配置 CircleCI 集成环境
- 编写参考文档
- Git workflow 配置

详见 Dev Guide wiki。

设计文档

所有的设计文档都保存在 Google Drive 中，其中包括以下资源：

- Technical Oversight Committee：ToC管理的文档
- Misc：一些杂项
- Working Groups：最重要的部分，各个工作组相关的设计文档
- Presentations：Istio 相关的演讲幻灯片，从这些文稿中可以快速了解 Istio
- Logo：Istio logo
- Eng：社区相关的维护文档

社区角色划分

根据对开发者和要求和贡献程度的不同，Istio 社区中包含以下角色：

- Collaborator：非正式贡献者，偶尔贡献，任何人都可以成为该角色
- Member：正式贡献者，经常贡献，必须有2个已有的 member 提名
- Approver：老手，可以批准 member 的贡献
- Lead：管理功能、项目和提议，必须由 ToC 提名
- Administrator：管理员，管理和控制权限，必须由 ToC 提名
- Vendor：贡献 Istio 项目的扩展

详见 Istio Community Roles。

各种功能的状态

Istio 中的所有 feature 根据是否生产可用、**API**兼容性、性能、维护策略分为三种状态：

- Alpha：仅仅可以作为 demo，无法生产上使用，也没有性能保证，随时都可能不维护
- Beta：可以在生产上使用了，也有版本化的 API 但是无法保证性能，保证三个月的维护
- Stable：可以上生产而且还能保证性能，API 向后兼容，保证一年的维护

Istio 的 feature 分为四大类：

- 流量管理：各种协议的支持、路由规则配置、Ingress TLS 等
- 可观察性：监控、日志、分布式追踪、服务依赖拓扑
- 安全性：各种 checker 和安全性配置
- Core：核心功能

功能划分与各种功能的状态详情请见：https://istio.io/about/feature-stages.html

Istio 教程

本文是 Istio 管理 Java 微服务的案例教程，使用的所有工具和软件全部基于开源方案，替换了 redhat-developer-demos/istio-tutorial 中的 minishift 环境，使用 kubernetes-vagrant-centos-cluster 替代，沿用了原有的微服务示例，使用 Zipkin 做分布式追踪而不是 Jaeger。

本文中的代码和 YAML 文件见 https://github.com/rootsongjc/istio-tutorial。

准备环境

在进行本教程前需要先准备以下工具和环境。

- 8G 以上内存
- Vagrant 2.0+
- Virtualbox 5.0 +
- 提前下载 kubernetes1.9.1 的 release 压缩包
- docker 1.12+
- kubectl 1.9.1+
- maven 3.5.2+
- istioctl 0.7.1
- git
- curl、gzip、tar
- kubetail
- siege

安装 Kubernetes

请参考 kubernetes-vagrant-centos-cluster 在本地启动拥有三个节点的 kubernetes 集群。

```
git clone https://github.com/rootsongjc/kubernetes-vagrant-cento
s-cluster.git
cd kubernetes-vagrant-centos-cluster
vagrant up
```

安装 Istio

在 kubernetes-vagrant-centos-cluster 中的包含 Istio 0.7.1 的安装 YAML 文件，运行下面的命令安装 Istio。

```
kubectl apply -f addon/istio/
```

运行示例

```
kubectl apply -n default -f <(istioctl kube-inject -f yaml/istio
-bookinfo/bookinfo.yaml)
```

在您自己的本地主机的 `/etc/hosts` 文件中增加如下配置项。

```
172.17.8.102 grafana.istio.jimmysong.io
172.17.8.102 servicegraph.istio.jimmysong.io
172.17.8.102 zipkin.istio.jimmysong.io
```

我们可以通过下面的URL地址访问以上的服务。

Service	URL
grafana	http://grafana.istio.jimmysong.io
servicegraph	http://servicegraph.istio.jimmysong.io/dotviz，http://servicegraph
zipkin	http://zipkin.istio.jimmysong.io

详细信息请参阅 https://istio.io/docs/guides/bookinfo.html

部署示例应用

在打包成镜像部署到 kubernetes 集群上运行之前，我们先在本地运行所有示例。

本教程中三个服务之间的依赖关系如下：

```
customer → preference → recommendation
```

`customer` 和 `preference` 微服务是基于 Spring Boot 构建的，`recommendation` 微服务是基于 vert.x 构建的。

`customer` 和 `preference` 微服务的 `pom.xml` 文件中都引入了 OpenTracing 和 Jeager 的依赖。

```
<dependency>
    <groupId>io.opentracing.contrib</groupId>
    <artifactId>opentracing-spring-cloud-starter</artifactId>
    <version>0.1.7</version>
</dependency>
<dependency>
    <groupId>com.uber.jaeger</groupId>
    <artifactId>jaeger-tracerresolver</artifactId>
    <version>0.25.0</version>
</dependency>
```

本地运行

我们首先在本地确定所有的微服务都可以正常运行，然后再打包镜像在 kubernetes 集群上运行。

启动 Jaeger

使用 docker 来运行 jagger。

```
docker run -d \
  --rm \
  -p5775:5775/udp \
  -p6831:6831/udp \
  -p6832:6832/udp \
  -p16686:16686 \
  -p14268:14268 \
  jaegertracing/all-in-one:1.3
```

Jaeger UI 地址 http://localhost:16686

Customer

```
cd customer/java/springboot
JAEGER_SERVICE_NAME=customer mvn \
  spring-boot:run \
  -Drun.arguments="--spring.config.location=src/main/resources/a
pplication-local.properties"
```

服务访问地址： http://localhost:8280

Preference

```
cd preference/java/springboot
JAEGER_SERVICE_NAME=preference mvn \
  spring-boot:run \
  -Drun.arguments="--spring.config.location=src/main/resources/a
pplication-local.properties"
```

服务访问地址：http://localhost:8180

Recommendation

```
cd recommendation/java/vertx
mvn vertx:run
```

服务访问地址：http://localhost:8080

所有服务都启动之后，此时访问 http://localhost:8280 将会看到如下输出。

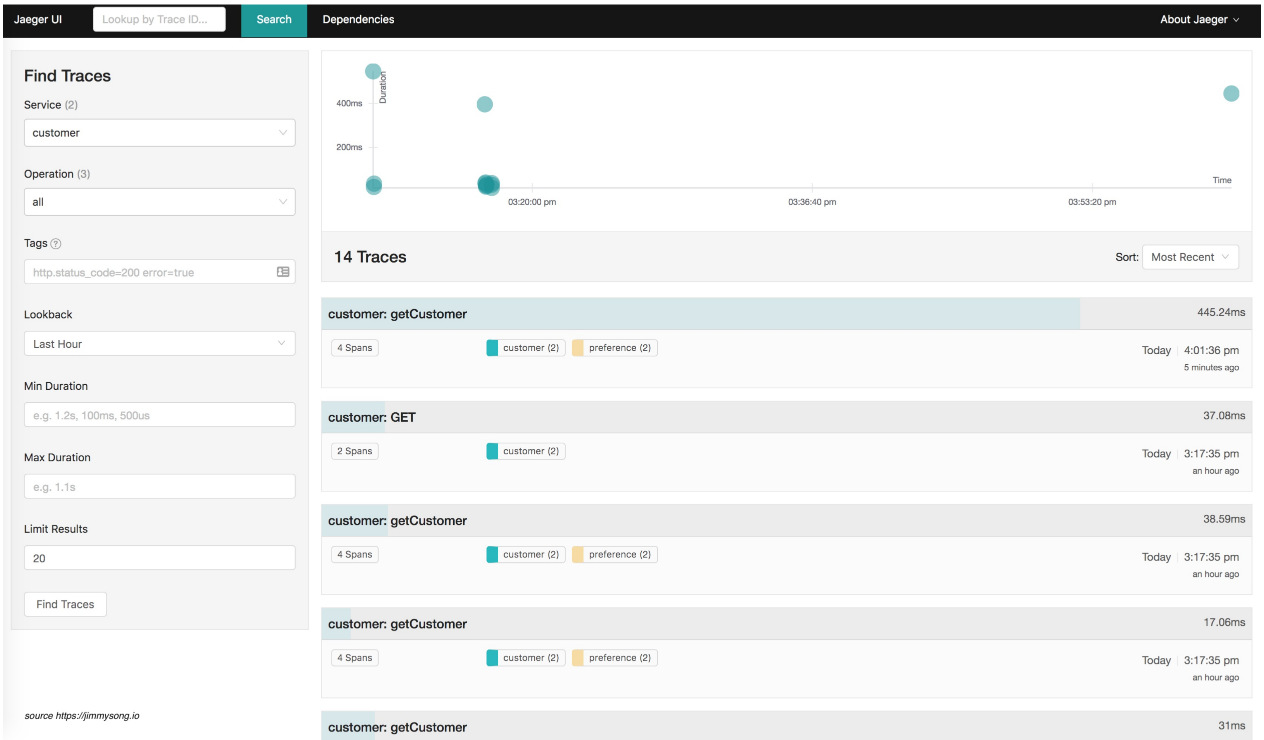
```
customer => preference => recommendation v1 from 'unknown': 1
```

每访问一次最后的数字就会加 1。

Jaeger

此时访问 http://localhost:16686 将看到 Jaeger query UI，所有应用将 metrics 发送到 Jeager 中。

可以在 Jaeger UI 中搜索 `customer` 和 `preference` service 的 trace 并查看每次请求的 tracing。



图片 - *Jaeger query UI*

构建镜像

在本地运行测试无误之后就可以构建镜像了。本教程中的容器镜像都是在 fabric8/java-jboss-openjdk8-jdk 的基础上构建的。只要将 Java 应用构建出 Jar 包然后放到 `/deployments` 目录下基础镜像就可以自动帮我们运行，所以我们看到着几个应用的 `Dockerfile` 文件中都没有执行入口，真正的执行入口是 run-java.sh。

Customer

构建 Customer 镜像。

```
cd customer/java/springboot
mvn clean package
docker build -t jimmysong/istio-tutorial-customer:v1 .
docker push jimmysong/istio-tutorial-customer:v1
```

第一次构建和上传需要花费一点时间，下一次构建就会很快。

Preference

构建 Preference 镜像。

```
cd preference/java/springboot
mvn clean package
docker build -t jimmysong/istio-tutorial-preference:v1 .
docker push jimmysong/istio-tutorial-preference:v1
```

Recommendation

构建 Recommendation 镜像。

```
cd recommendation/java/vertx
mvn clean package
docker build -t jimmysong/istio-tutorial-recommendation:v1 .
docker push jimmysong/istio-tutorial-recommendation:v1
```

现在三个 docker 镜像都构建完成了，我们检查一下。

```
$ docker images | grep istio-tutorial
REPOSITORY                                TAG                 IM
AGE ID          CREATED             SIZE
jimmysong/istio-tutorial-recommendation   v1                  d3
1dd858c300      51 seconds ago      443MB
jimmysong/istio-tutorial-preference       v1                  e5
f0be361477      6 minutes ago       459MB
jimmysong/istio-tutorial-customer         v1                  d9
601692673e      13 minutes ago      459MB
```

部署到 Kubernetes

使用下面的命令将以上服务部署到 kubernetes。

```
# create new namespace
kubectl create ns istio-tutorial

# deploy recommendation
kubectl apply -f <(istioctl kube-inject -f recommendation/kubern
etes/Deployment.yml) -n istio-tutorial
kubectl apply -f recommendation/kubernetes/Service.yml

# deploy preferrence
kubectl apply -f <(istioctl kube-inject -f preference/kubernetes
/Deployment.yml) -n istio-tutorial
kubectl apply -f preference/kubernetes/Service.yml

# deploy customer
kubectl apply -f <(istioctl kube-inject -f customer/kubernetes/D
eployment.yml) -n istio-tutorial
kubectl apply -f customer/kubernetes/Service.yml
```

注意：`preference` 和 `customer` 应用启动速度比较慢，我们将 livenessProb 配置中的 `initialDelaySeconds` 设置为 **20** 秒。

查看 Pod 启动状态：

```
kubectl get pod -w -n istio-tutorial
```

增加 Ingress 配置

为了在 kubernetes 集群外部访问 customer 服务，我们需要增加 ingress 配置。

```
kubectl apply -f ingress/ingress.yaml
```

修改本地的 `/etc/hosts` 文件，增加一条配置。

```
172.17.8.102 customer.istio-tutorial.jimmysong.io
```

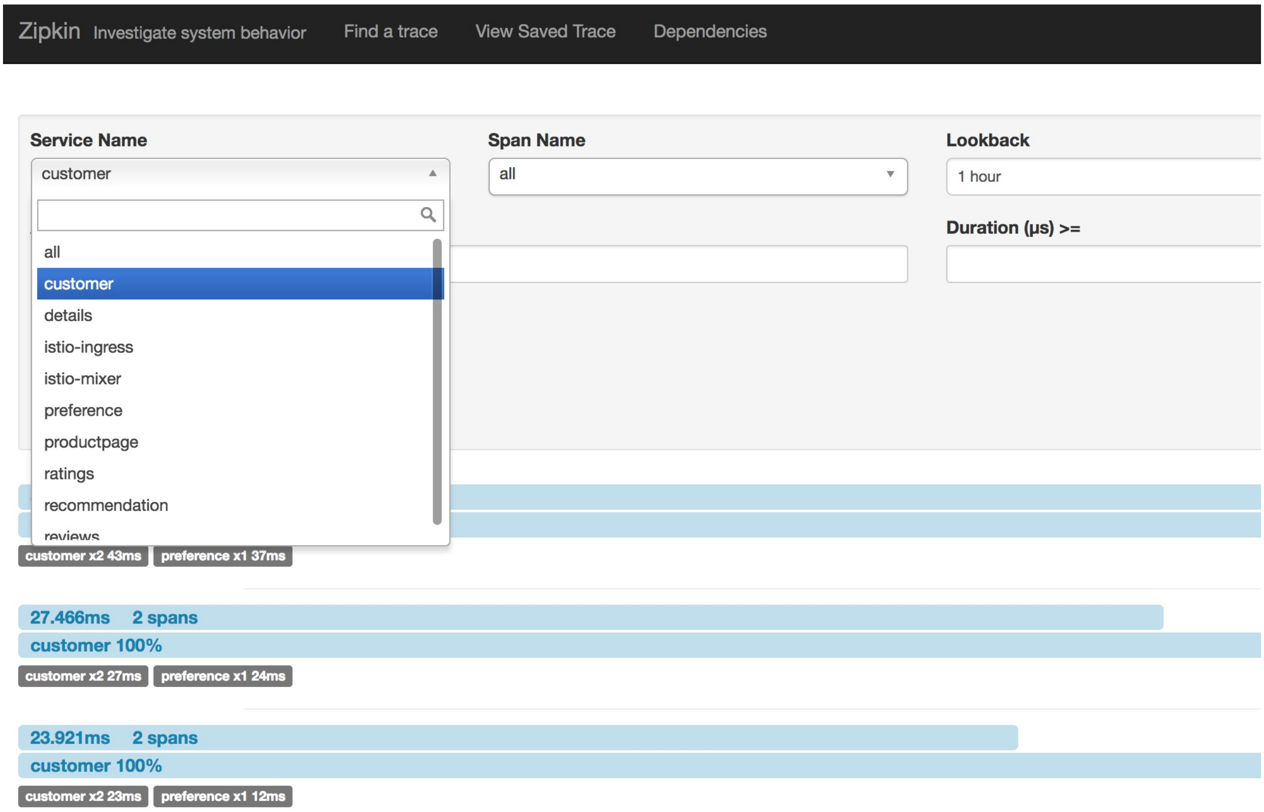
现在访问 http://customer.istio-tutorial.jimmysong.io 将看到如下输出：

```
customer => preference => recommendation v1 from '6fc97476f8-m2n
tp': 1
```

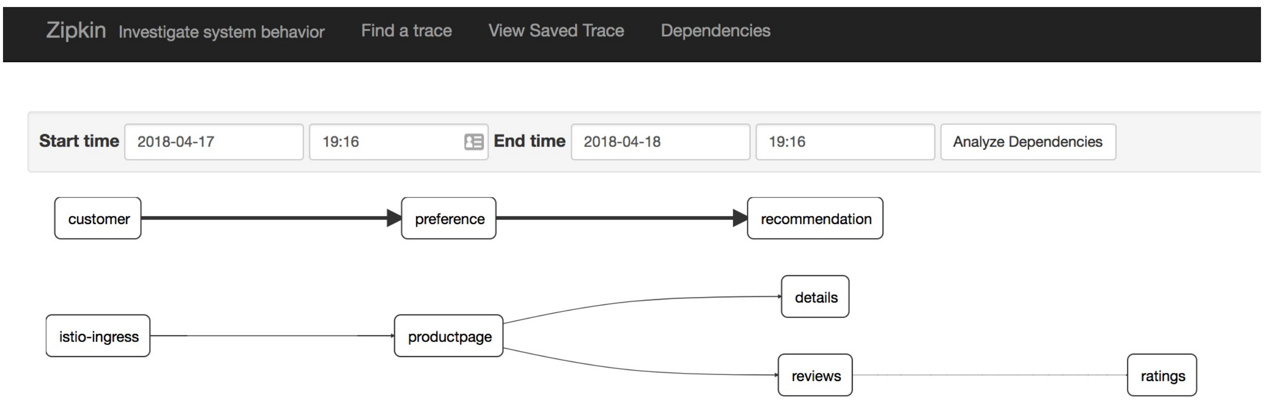
批量访问该地址。

```
./bin/poll_customer.sh
```

访问 http://servicegraph.istio.jimmysong.io/dotviz 查看服务的分布式追踪和依赖关系。

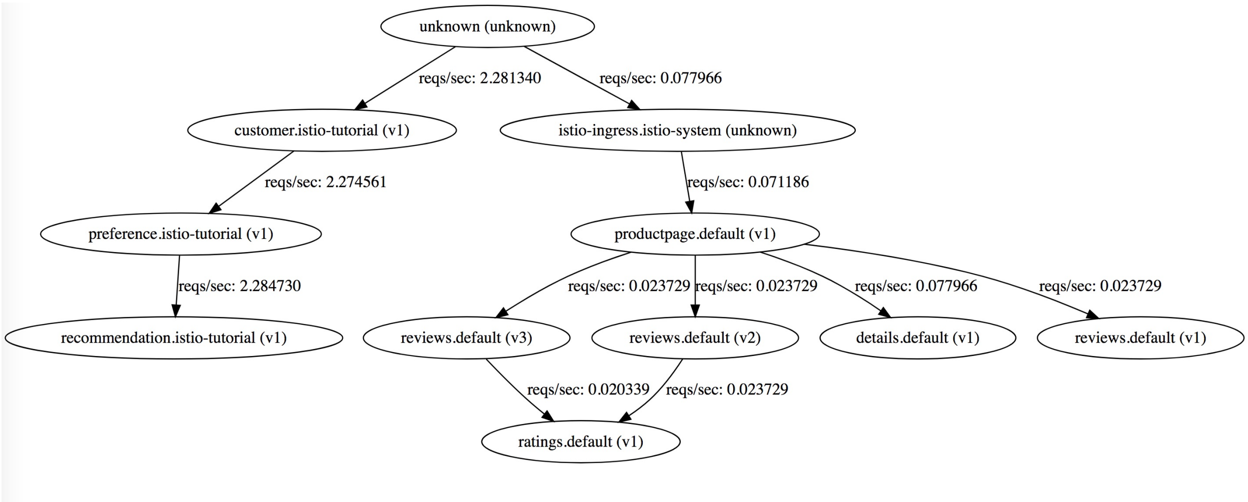


图片 - 分布式追踪



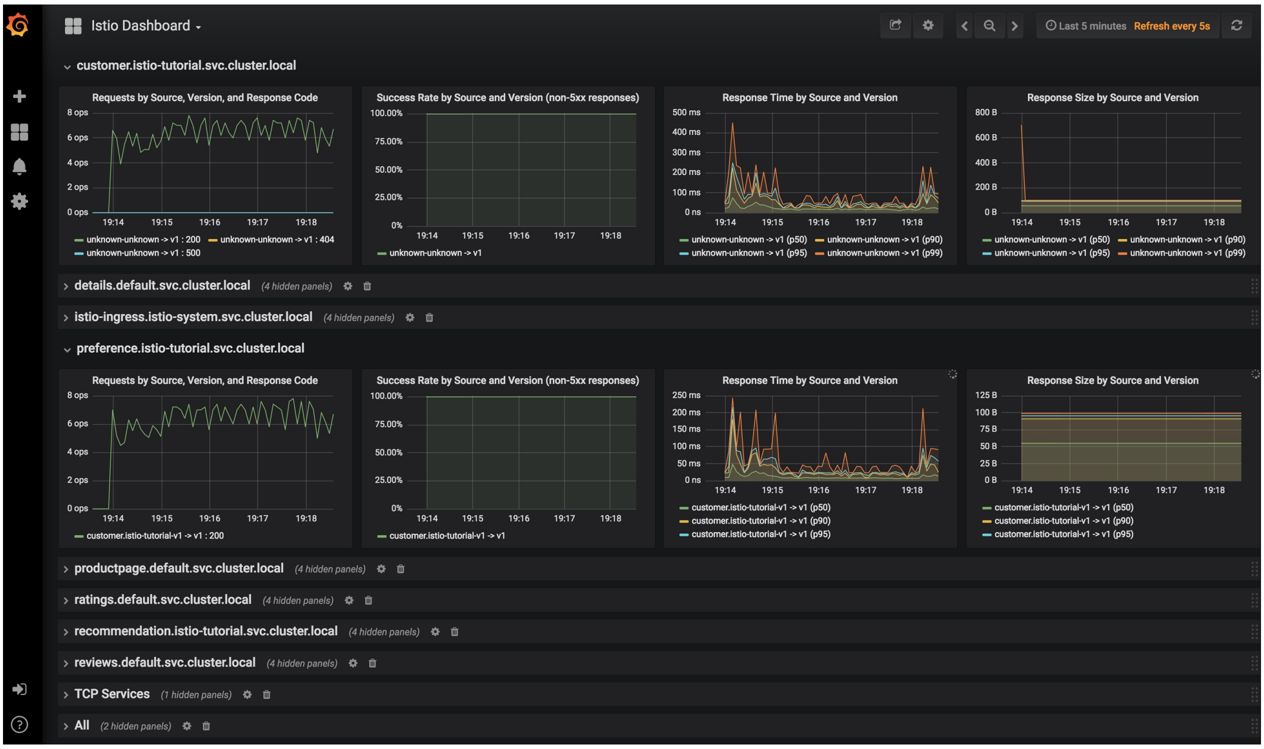
图片 - 依赖关系

访问 http://servicegraph.istio.jimmysong.io/dotviz 查看服务间的关系图和 QPS。



图片 - 服务关系图和*QPS*

访问 http://grafana.istio.jimmysong.io 查看 Service Mesh 的监控信息。



图片 - *Grafana* 监控

Istio 使用示例

为了试用 Istio 中的各种功能，我们需要为应用构建多个版本，我们为 recommendation 构建 v2 版本的镜像，看看如何使用 Istio 控制微服务的流量。

构建 recommendation:v2

我们将构建新版的 `recommendation` 服务的镜像，并观察 `customer` 对不同版本的 `recommendataion` 服务的访问频率。

修改 `recommendation/java/vertx/src/main/java/com/redhat/developer/demos/recommendation/RecommendationVerticle.java` 程序中代码。

将 `private static final String RESPONSE_STRING_FORMAT = "recommendation v1 from '%s': %d\n";` 修改为 `private static final String RESPONSE_STRING_FORMAT = "recommendation v2 from '%s': %d\n";`

并构建 `recommendation:v2` 镜像。

```
cd recommendation/java/vertx
mvn clean package
docker build -t jimmysong/istio-tutorial-recommendation:v2 .
docker push jimmysong/istio-tutorial-recommendation:v2
```

将应用部署到 kubernetes。

```
# deploy recommendation
kubectl apply -f <(istioctl kube-inject -f recommendation/kubern
etes/Deployment-v2.yml) -n istio-tutorial
```

现在再访问 `customer` 服务，将看到如下输出：

```
$ bin/poll_customer.sh
customer => preference => recommendation v2 from '77b9f6cc68-5xs
27': 1
customer => preference => recommendation v1 from '6fc97476f8-m2n
tp': 3581
customer => preference => recommendation v2 from '77b9f6cc68-5xs
27': 2
customer => preference => recommendation v1 from '6fc97476f8-m2n
tp': 3582
customer => preference => recommendation v2 from '77b9f6cc68-5xs
27': 3
customer => preference => recommendation v1 from '6fc97476f8-m2n
tp': 3583
customer => preference => recommendation v2 from '77b9f6cc68-5xs
27': 4
```

我们可以看到 v1 和 v2 版本的 `recommendation` 服务会被间隔访问到。

我们再将 v2 版本的 `recommendation` 实例数设置成 2 个。

```
kubectl scale --replicas=2 deployment/recommendation-v2 -n istio
-tutorial
kubectl get pod -w -n istio-tutorial
```

观察 `recommendation-v2` Pod 达到两个之后再访问 `customer` 服务。

```
$ bin/poll_customer.sh
customer => preference => recommendation v2 from '77b9f6cc68-j9f
gj': 1
customer => preference => recommendation v2 from '77b9f6cc68-5xs
27': 71
customer => preference => recommendation v1 from '6fc97476f8-m2n
tp': 3651
customer => preference => recommendation v2 from '77b9f6cc68-j9f
gj': 2
customer => preference => recommendation v2 from '77b9f6cc68-5xs
27': 72
customer => preference => recommendation v1 from '6fc97476f8-m2n
tp': 3652
customer => preference => recommendation v2 from '77b9f6cc68-j9f
gj': 3
customer => preference => recommendation v2 from '77b9f6cc68-5xs
27': 73
customer => preference => recommendation v1 from '6fc97476f8-m2n
tp': 3653
```

观察输出中 v1 和 v2 版本 `recommendation` 的访问频率。

将 `recommendataion` 服务的实例数恢复为 1。

```
kubectl scale --replicas=1 deployment/recommendation-v2
```

修改 Istio RouteRules

以下所有路有规则都是针对 `recommendation` 服务，并在 repo 的根目录下执行。

将所有流量打给 **v2**

下面将演示如何动态的划分不同版本服务间的流量，将所有的流量都打到 `recommendation:v2` 。

```
istioctl create -f istiofiles/route-rule-recommendation-v2.yml -
n istio-tutorial
```

现在再访问 `customer` 服务将看到所有的流量都会打到 `recommendation:v2` 。

删除 RouteRules 后再访问 `customer` 服务将看到又恢复了 v1 和 v2 版本的 `recommendation` 服务的间隔访问。

```
istioctl delete routerule recommendation-default
```

切分流量

将 90% 的流量给 v1，10% 的流量给 v2。

```
istioctl create -f istiofiles/route-rule-recommendation-v1_and_v
2.yml -n istio-tutorial
```

执行 `bin/poll_customer.sh` 观察访问情况。

要想动态切分流量只要修改 RouteRules 中的 `weight` 配置即可。

```
apiVersion: config.istio.io/v1alpha2
kind: RouteRule
metadata:
  name: recommendation-v1-v2
spec:
  destination:
    namespace: istio-tutorial
    name: recommendation
  precedence: 5
  route:
  - labels:
      version: v1
    weight: 90
  - labels:
      version: v2
    weight: 10
```

因为 RouteRule 有优先级，为了继续后面的实验，在验证完成后删除该 RouteRule。

```
istioctl delete routerule recommendation-v1-v2 -n istio-tutorial
```

故障注入

有时候我们为了增强系统的健壮性，需要对系统做混沌工程，故意注入故障，并保障服务可以自动处理这些故障。

注入 **HTTP 503** 错误

```
istioctl create -f istiofiles/route-rule-recommendation-503.yml
-n istio-tutorial
```

有 50% 的几率报 503 错误。

```
$ bin/poll_customer.sh
customer => preference => recommendation v2 from '77b9f6cc68-5xs
27': 135
customer => 503 preference => 503 fault filter abort
customer => preference => recommendation v1 from '6fc97476f8-m2n
tp': 3860
customer => 503 preference => 503 fault filter abort
customer => 503 preference => 503 fault filter abort
customer => preference => recommendation v2 from '77b9f6cc68-5xs
27': 136
customer => preference => recommendation v1 from '6fc97476f8-m2n
tp': 3861
customer => 503 preference => 503 fault filter abort
customer => 503 preference => 503 fault filter abort
customer => preference => recommendation v2 from '77b9f6cc68-5xs
27': 137
customer => 503 preference => 503 fault filter abort
```

清理 RouteRule。

```
istioctl delete routerule recommendation-503 -n istio-tutorial
```

增加延迟

增加服务的访问延迟。

```
istioctl create -f istiofiles/route-rule-recommendation-delay.ym
l -n istio-tutorial
```

会有 50% 的几率访问 `recommendation` 服务有 7 秒的延迟。百分比和延迟时间可以在 RouteRule 中配置。

清理 RouteRule。

```
istioctl delete routerule recommendation-delay -n istio-tutorial
```

重试

让服务不是直接失败，而是增加重试机制。

我们下面将同时应用两条 RouteRule，让访问 `recommendation` 服务时有 50% 的几率出现 503 错误，并在出现错误的时候尝试访问 v2 版本，超时时间为 2 秒。

```
istioctl create -f istiofiles/route-rule-recommendation-v2_503.y
ml -n istio-tutorial
istioctl create -f istiofiles/route-rule-recommendation-v2_retry
.yml -n istio-tutorial
```

执行 `bin/poll_customer.sh` 我们看到一开始有些 503 错误，然后所有的流量都流向了 v2。

清理 RouteRules。

```
istioctl delete routerule recommendation-v2-retry -n istio-tutor
ial
istioctl delete routerule recommendation-v2-503 -n istio-tutoria
l
```

超时

设置超时时间，只有服务访问超时才认定服务访问失败。

取消注释 `recommendation/java/vertx/src/main/java/com/redhat/developer/demos/recommendation/RecommendationVerticle.java` 中的下面一行，增加超时时间为 3 秒。

```
router.get("/").handler(this::timeout);
```

重新生成镜像。

```
cd recommendation/java/vertx
mvn clean package
docker build -t jimmysong/istio-tutorial-recommendation:v2 .
docker push jimmysong/istio-tutorial-recommendation:v2
```

重新部署到 kubernetes。

```
kubectl delete -f recommendation/kubernetes/Deployment-v2.yml
```

因为我们重新构建的镜像使用了同样的名字和 tag，而之前在 `Deployment-v2.yml` 中配置的镜像拉取策略是 `IfNotPresent` ，这样的话即使我们构建了新的镜像也无法应用到集群上，因此将镜像拉取策略改成 `Always` 确保每次启动 Pod 的时候都会拉取镜像。

```
kubectl apply -f <(istioctl kube-inject -f recommendation/kubern
etes/Deployment-v2.yml) -n istio-tutorial
```

启用超时 RouteRules。

```
istioctl create -f istiofiles/route-rule-recommendation-timeout.
yml -n istio-tutorial
```

访问 `customer` 服务将看到如下输出：

```
$ bin/poll_customer.sh
customer => 503 preference => 504 upstream request timeout
customer => preference => recommendation v1 from '6fc97476f8-m2n
tp': 4002
customer => 503 preference => 504 upstream request timeout
customer => preference => recommendation v1 from '6fc97476f8-m2n
tp': 4003
customer => 503 preference => 504 upstream request timeout
customer => preference => recommendation v1 from '6fc97476f8-m2n
tp': 4004
```

清理 RouteRules。

```
istioctl delete routerule recommendation-timeout -n istio-tutori
al
```

基于 user-agent 的智能路由（金丝雀发布）

User-agent 是一个字符串，其中包含了浏览器的信息，访问 https://www.whoishostingthis.com/tools/user-agent 获取你的 user-agent。

我的 user-agent 是：

```
Mozilla/5.0 (Macintosh; Intel Mac OS X 10_13_4) AppleWebKit/537.
36 (KHTML, like Gecko) Chrome/65.0.3325.181 Safari/537.36
```

将所有的流量打到 v1。

```
istioctl create -f istiofiles/route-rule-recommendation-v1.yml -
n istio-tutorial
```

将使用 Safari 浏览器访问的流量打到 v2。

```
istioctl create -f istiofiles/route-rule-safari-recommendation-v
2.yml -n istio-tutorial
```

谁用 Safari 或者 Chrome（Chrome 浏览器的 user-agent 中也包含 Safari 字段）访问 http://customer.istio-tutorial.jimmysong.io/ 在经过 3 秒钟（我们在前面重新编译 v2 镜像，设置了 3 秒超时时间）后将看到访问 v2 的输出。

或者使用 curl 访问。

```
curl -A Safari http://customer.istio-tutorial.jimmysong.io/
curl -A Firefox http://customer.istio-tutorial.jimmysong.io/
```

观察返回的结果。

将移动端用户的流量导到 v2。

```
istioctl create -f istiofiles/route-rule-mobile-recommendation-v
2.yml -n istio-tutorial

curl -A "Mozilla/5.0 (iPhone; U; CPU iPhone OS 4(KHTML, like Gec
ko) Version/5.0.2 Mobile/8J2 Safari/6533.18.5" http://customer.i
stio-tutorial.jimmysong.io/
```

观察输出的结果。

清理 RouteRules。

```
istioctl delete routerule recommendation-mobile -n istio-tutoria
l
istioctl delete routerule recommendation-safari -n istio-tutoria
l
istioctl delete routerule recommendation-default -n istio-tutori
al
```

镜像流量

确保当前至少运行了两个版本的 `recommendation` 服务，并且没有 RouteRule。

注：可以使用 `istioctl get routerule` 获取 RouteRule。

设置流量镜像，将所有 v1 的流量都被镜像到 v2。

```
istioctl create -f istiofiles/route-rule-recommendation-v1-mirro
r-v2.yml -n istio-tutorial
bin/poll_customer.sh
```

查看 recommendation-v2 的日志。

```
kubectl logs -f `oc get pods|grep recommendation-v2|awk '{ print
$1 }'` -c recommendation
```

访问控制

Istio 可以设置服务访问的黑白名单，如果没有权限的话会返回 HTTP 404 Not Found。

白名单

```
istioctl create -f istiofiles/acl-whitelist.yml -n istio-tutoria
l
```

此时访问 `customer` 服务。

```
$ bin/poll_customer.sh
customer => 404 NOT_FOUND:preferencewhitelist.listchecker.istio-
tutorial:customer is not whitelisted
```

重置环境。

```
istioctl delete -f istiofiles/acl-whitelist.yml -n istio-tutoria
l
```

黑名单

设置黑名单，所有位于黑名单中的流量将获得 403 Forbidden 返回码。

```
istioctl create -f istiofiles/acl-blacklist.yml -n istio-tutoria
l
```

此时访问 `customer` 服务。

```
$ bin/poll_customer.sh
customer => 403 PERMISSION_DENIED:denycustomerhandler.denier.ist
io-tutorial:Not allowed
```

重置环境。

```
istioctl delete -f istiofiles/acl-blacklist.yml -n istio-tutoria
l
```

负载均衡

Kubernetes 中默认的负载均衡策略是 round-robin，当然我们可以使用 Istio 把它修改成 random。

增加 v1 的实例数。

```
kubectl scale deployment recommendation-v1 --replicas=2 -n istio
-tutorial
```

持续访问 `customer` 服务。

```
bin/poll_customer.sh
```

保持前台输出，观察流量的行为。

应用负载均衡策略。

```
istioctl create -f istiofiles/recommendation_lb_policy_app.yml -
n istio-tutorial
```

观察一段时间流量的行为后，重置环境。

```
istioctl delete -f istiofiles/recommendation_lb_policy_app.yml -
n istio-tutorial
kubectl scale deployment recommendation-v1 --replicas=1 -n istio
-tutorial
```

速率限制

暂时不可用

断路器

当达到最大连接数和最大挂起请求数时快速失败。

将流量在 v1 和 v2 之间均分。

```
istioctl create -f istiofiles/route-rule-recommendation-v1_and_v
2_50_50.yml -n istio-tutorial
```

未开启断路器的时候启动负载测试。

```
$ siege -r 2 -c 20 -v customer.istio-tutorial.jimmysong.io
New configuration template added to /Users/jimmysong/.siege
Run siege -C to view the current settings in that file
** SIEGE 4.0.4
** Preparing 20 concurrent users for battle.
The server is now under siege...
HTTP/1.1 200     0.10 secs:      75 bytes ==> GET  /
HTTP/1.1 200     0.12 secs:      75 bytes ==> GET  /
HTTP/1.1 200     0.13 secs:      75 bytes ==> GET  /
HTTP/1.1 200     0.13 secs:      75 bytes ==> GET  /
HTTP/1.1 200     0.13 secs:      75 bytes ==> GET  /
HTTP/1.1 200     0.17 secs:      75 bytes ==> GET  /
HTTP/1.1 200     3.12 secs:      74 bytes ==> GET  /
HTTP/1.1 200     3.14 secs:      75 bytes ==> GET  /
HTTP/1.1 200     3.15 secs:      74 bytes ==> GET  /
HTTP/1.1 200     3.15 secs:      74 bytes ==> GET  /
HTTP/1.1 200     3.17 secs:      75 bytes ==> GET  /
HTTP/1.1 200     3.17 secs:      75 bytes ==> GET  /
HTTP/1.1 200     3.20 secs:      75 bytes ==> GET  /
HTTP/1.1 200     3.20 secs:      74 bytes ==> GET  /
HTTP/1.1 200     0.05 secs:      75 bytes ==> GET  /
HTTP/1.1 200     0.12 secs:      75 bytes ==> GET  /
```

```
HTTP/1.1 200     3.15 secs:      75 bytes ==> GET  /
HTTP/1.1 200     3.25 secs:      75 bytes ==> GET  /
HTTP/1.1 200     3.26 secs:      75 bytes ==> GET  /
HTTP/1.1 200     3.14 secs:      75 bytes ==> GET  /
HTTP/1.1 200     3.58 secs:      74 bytes ==> GET  /
HTTP/1.1 200     6.15 secs:      74 bytes ==> GET  /
HTTP/1.1 200     6.16 secs:      75 bytes ==> GET  /
HTTP/1.1 200     3.03 secs:      74 bytes ==> GET  /
HTTP/1.1 200     6.06 secs:      75 bytes ==> GET  /
HTTP/1.1 200     6.04 secs:      75 bytes ==> GET  /
HTTP/1.1 200     3.11 secs:      74 bytes ==> GET  /
HTTP/1.1 200     3.09 secs:      75 bytes ==> GET  /
HTTP/1.1 200     6.15 secs:      74 bytes ==> GET  /
HTTP/1.1 200     6.71 secs:      74 bytes ==> GET  /
HTTP/1.1 200     3.52 secs:      75 bytes ==> GET  /
^C
Lifting the server siege...
Transactions:                     31 hits
Availability:                 100.00 %
Elapsed time:                   7.99 secs
Data transferred:               0.00 MB
Response time:                   2.99 secs
Transaction rate:               3.88 trans/sec
Throughput:                   0.00 MB/sec
Concurrency:                  11.60
Successful transactions:           31
Failed transactions:                0
Longest transaction:             6.71
Shortest transaction:             0.05
```

所有的请求都成功了，但是性能很差，因为 v2 版本设置了 3 秒的超时时间。

我们启用下断路器。

```
istioctl create -f istiofiles/recommendation_cb_policy_version_v
2.yml -n istio-tutorial
```

重新测试一下。

```
$ siege -r 2 -c 20 -v customer.istio-tutorial.jimmysong.io
** SIEGE 4.0.4
** Preparing 20 concurrent users for battle.
The server is now under siege...
HTTP/1.1 200     0.07 secs:      75 bytes ==> GET  /
HTTP/1.1 503     0.07 secs:      92 bytes ==> GET  /
HTTP/1.1 200     0.07 secs:      75 bytes ==> GET  /
HTTP/1.1 503     0.12 secs:      92 bytes ==> GET  /
HTTP/1.1 503     0.12 secs:      92 bytes ==> GET  /
HTTP/1.1 200     0.16 secs:      75 bytes ==> GET  /
```

```
HTTP/1.1 503     0.16 secs:      92 bytes ==> GET  /
HTTP/1.1 503     0.21 secs:      92 bytes ==> GET  /
HTTP/1.1 503     0.21 secs:      92 bytes ==> GET  /
HTTP/1.1 200     0.24 secs:      75 bytes ==> GET  /
HTTP/1.1 200     0.24 secs:      75 bytes ==> GET  /
HTTP/1.1 503     0.14 secs:      92 bytes ==> GET  /
HTTP/1.1 503     0.29 secs:      92 bytes ==> GET  /
HTTP/1.1 503     0.13 secs:      92 bytes ==> GET  /
HTTP/1.1 503     0.18 secs:      92 bytes ==> GET  /
HTTP/1.1 503     0.13 secs:      92 bytes ==> GET  /
HTTP/1.1 200     0.11 secs:      75 bytes ==> GET  /
HTTP/1.1 200     0.39 secs:      75 bytes ==> GET  /
HTTP/1.1 200     0.24 secs:      75 bytes ==> GET  /
HTTP/1.1 503     0.44 secs:      92 bytes ==> GET  /
HTTP/1.1 200     0.43 secs:      75 bytes ==> GET  /
HTTP/1.1 200     0.44 secs:      75 bytes ==> GET  /
HTTP/1.1 503     0.40 secs:      92 bytes ==> GET  /
HTTP/1.1 200     0.47 secs:      75 bytes ==> GET  /
HTTP/1.1 503     0.42 secs:      92 bytes ==> GET  /
HTTP/1.1 200     0.42 secs:      75 bytes ==> GET  /
HTTP/1.1 200     0.06 secs:      75 bytes ==> GET  /
HTTP/1.1 503     0.07 secs:      92 bytes ==> GET  /
HTTP/1.1 200     0.15 secs:      75 bytes ==> GET  /
HTTP/1.1 200     0.12 secs:      75 bytes ==> GET  /
HTTP/1.1 503     0.57 secs:      92 bytes ==> GET  /
HTTP/1.1 503     0.18 secs:      92 bytes ==> GET  /
HTTP/1.1 503     0.52 secs:      92 bytes ==> GET  /
HTTP/1.1 503     0.65 secs:      92 bytes ==> GET  /
HTTP/1.1 503     0.42 secs:      92 bytes ==> GET  /
HTTP/1.1 200     0.09 secs:      75 bytes ==> GET  /
HTTP/1.1 200     0.43 secs:      75 bytes ==> GET  /
HTTP/1.1 503     0.04 secs:      92 bytes ==> GET  /
HTTP/1.1 200     4.15 secs:      74 bytes ==> GET  /
HTTP/1.1 200     0.01 secs:      75 bytes ==> GET  /

Transactions:                     19 hits
Availability:                  47.50 %
Elapsed time:                   4.16 secs
Data transferred:               0.00 MB
Response time:                   0.72 secs
Transaction rate:               4.57 trans/sec
Throughput:                   0.00 MB/sec
Concurrency:                   3.31
Successful transactions:              19
Failed transactions:                  21
Longest transaction:                4.15
Shortest transaction:                0.01
```

我们可以看到在启用了断路器后各项性能都有提高。

清理配置。

```
istioctl delete routerule recommendation-v1-v2 -n istio-tutorial
istioctl delete -f istiofiles/recommendation_cb_policy_version_v
2.yml -n istio-tutorial
```

Pool Ejection

所谓的 Pool Ejection 就是当某些实例出现错误（如返回 5xx 错误码）临时将该实例弹出一段时间后（窗口期，可配置），然后再将其加入到负载均衡池中。我们的例子中配置的窗口期是 15 秒。

将 v1 和 v2 的流量均分。

```
istioctl create -f istiofiles/route-rule-recommendation-v1_and_v
2_50_50.yml -n istio-tutorial
```

增加 v2 的实例个数。

```
kubectl scale deployment recommendation-v2 --replicas=2 -n istio
-tutorial
kubectl get pods -w
```

等待所有的 Pod 的状态都启动完成。

现在到 v2 的容器中操作。

```
$ kubectl exec recommendation-v2-785465d9cd-225ms -c recommendat
ion /bin/bash
$ curl localhost:8080/misbehave
Following requests to '/' will return a 503
```

增加 Pool Ejection 配置。

```
istioctl create -f istiofiles/recommendation_cb_policy_pool_ejec
tion.yml -n istio-tutorial
```

此时再访问 `customer` 服务。

```
$ bin/poll_customer.sh
customer => preference => recommendation v1 from '6fc97476f8-m2n
tp': 10505
customer => preference => recommendation v2 from '785465d9cd-225
ms': 2407
customer => preference => recommendation v1 from '6fc97476f8-m2n
tp': 10506
customer => preference => recommendation v2 from '785465d9cd-225
ms': 2408
customer => preference => recommendation v1 from '6fc97476f8-m2n
tp': 10507
customer => preference => recommendation v1 from '6fc97476f8-m2n
tp': 10508
customer => preference => recommendation v1 from '6fc97476f8-m2n
tp': 10509
customer => 503 preference => 503 recommendation misbehavior fro
m '785465d9cd-ldc6j'
customer => preference => recommendation v2 from '785465d9cd-225
ms': 2409
customer => preference => recommendation v2 from '785465d9cd-225
ms': 2410
```

我们看到窗口期生效了，当出现 503 错误后至少 15 秒后才会出现第二次。

即使有了负载均衡池弹出策略对于系统的弹性来说依然还不够，如果你的服务有多个可用实例，可以将断路器、重试、**Pool Ejection** 等策略组合起来使用。

例如在以上的 Pool Ejection 的基础上增加重试策略。

```
istioctl replace -f istiofiles/route-rule-recommendation-v1_and_
v2_retry.yml -n istio-tutorial
```

现在再访问 `customer` 服务就看不到 503 错误了。

清理配置。

```
kubectl scale deployment recommendation-v2 --replicas=1 -n istio
-tutorial
istioctl delete routerule recommendation-v1-v2 -n istio-tutorial
istioctl delete -f istiofiles/recommendation_cb_policy_pool_ejec
tion.yml -n istio-tutorial
```

Egress

Egress 是用来配置 Istio serivce mesh 中的服务对外部服务的访问策略。

具体配置请参考 控制 Egress 流量。

以下示例还有问题，无法正常工作。

构建示例镜像 egresshttpbin。

```
cd egress/egresshttpbin/
mvn clean package
docker build -t jimmysong/istio-tutorial-egresshttpbin:v1 .
docker push jimmysong/istio-tutorial-egresshttpbin:v1
```

部署到 Kubernetes。

```
kubectl apply -f <(istioctl kube-inject -f egress/egresshttpbin/
src/main/kubernetes/Deployment.yml) -n istio-toturial
kubectl create -f egress/egresshttpbin/src/main/kubernetes/Servi
ce.yml
```

为了在 kubernetes 集群外部访问到该服务，修改增加 ingress 配置并修改本地的 `/etc/hosts` 文件，我们在前面已经完成了，此处不再赘述。

构建示例镜像 egressgithub。

```
cd egress/egressgithub
mvn clean package
docker build -t jimmysong/istio-tutorial-egressgithub:v1 .
docker push jimmysong/istio-tutorial-egressgithub:v1
```

部署到 Kubernetes。

```
kubectl apply -f <(istioctl kube-inject -f egress/egressgithub/s
rc/main/kubernetes/Deployment.yml) -n istio-tutorial
kubectl create -f egress/egressgithub/src/main/kubernetes/Servic
e.yml
```

增加 Egress 配置。

```
istioctl create -f istiofiles/egress_httpbin.yml -n istio-tutori
al
```

到 egresshttpbin 容器中测试。

```
kubectl exec -it $(oc get pods -o jsonpath="{.items[*].metadata.
name}" -l app=egresshttpbin,version=v1) -c egresshttpbin /bin/ba
sh

curl localhost:8080

curl httpbin.org/user-agent

curl httpbin.org/headers

exit
```

增加对 jimmysong.io 的 egress 配置。

```
cat <<EOF | istioctl create -f -
apiVersion: config.istio.io/v1alpha2
kind: EgressRule
metadata:
  name: jimmysong-egress-rule
  namespace: istio-tutorial
spec:
  destination:
    service: jimmysong.io
  ports:
    - port: 443
      protocol: https
EOF
```

增加 Egress 配置。

```
istioctl create -f istiofiles/egress_github.yml -n istio-tutoria
l
```

到 egressgithub 容器中测试。

```
kubectl exec -it $(oc get pods -o jsonpath="{.items[*].metadata.
name}" -l app=egressgithub,version=v1) -c egressgithub /bin/bash

curl http://jimmysong:443

exit
```

清理环境。

```
istioctl delete egressrule httpbin-egress-rule jimmysong-egress-
rule github-egress-rule -n istio-tutorial
```

参考

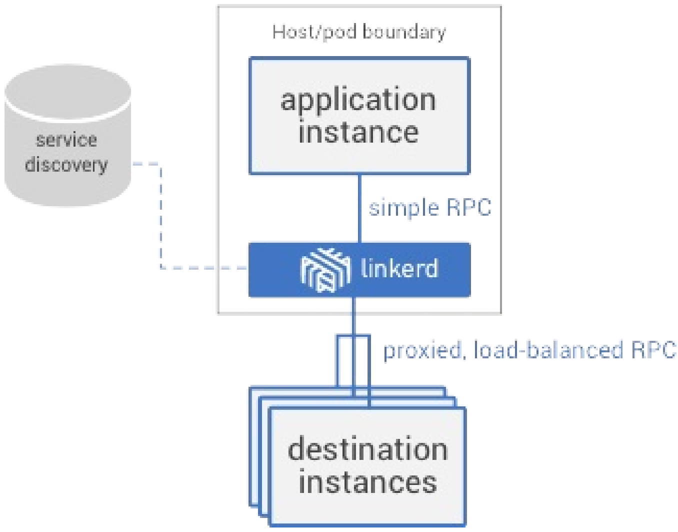
- https://github.com/redhat-developer-demos/istio-tutorial
- Book - Introducing Istio Service Mesh for Microservices

Linkerd简介

Linkerd是一个用于云原生应用的开源、可扩展的service mesh（一般翻译成服务网格，还有一种说法叫"服务啮合层“，见Istio：用于微服务的服务啮合层）。

Linkerd是什么

Linkerd的出现是为了解决像twitter、google这类超大规模生产系统的复杂性问题。Linkerd不是通过控制服务之间的通信机制来解决这个问题，而是通过在服务实例之上添加一个抽象层来解决的。



图片 - *source https://linkerd.io*

Linkerd负责跨服务通信中最困难、易出错的部分，包括延迟感知、负载平衡、连接池、TLS、仪表盘、请求路由等——这些都会影响应用程序伸缩性、性能和弹性。

如何运行

Linkerd作为独立代理运行，无需特定的语言和库支持。应用程序通常会在已知位置运行linkerd实例，然后通过这些实例代理服务调用——即不是直接连接到目标服务，服务连接到它们对应的linkerd实例，并将它们视为目标服务。

在该层上，linkerd应用路由规则，与现有服务发现机制通信，对目标实例做负载均衡——与此同时调整通信并报告指标。

通过延迟调用linkerd的机制，应用程序代码与以下内容解耦：

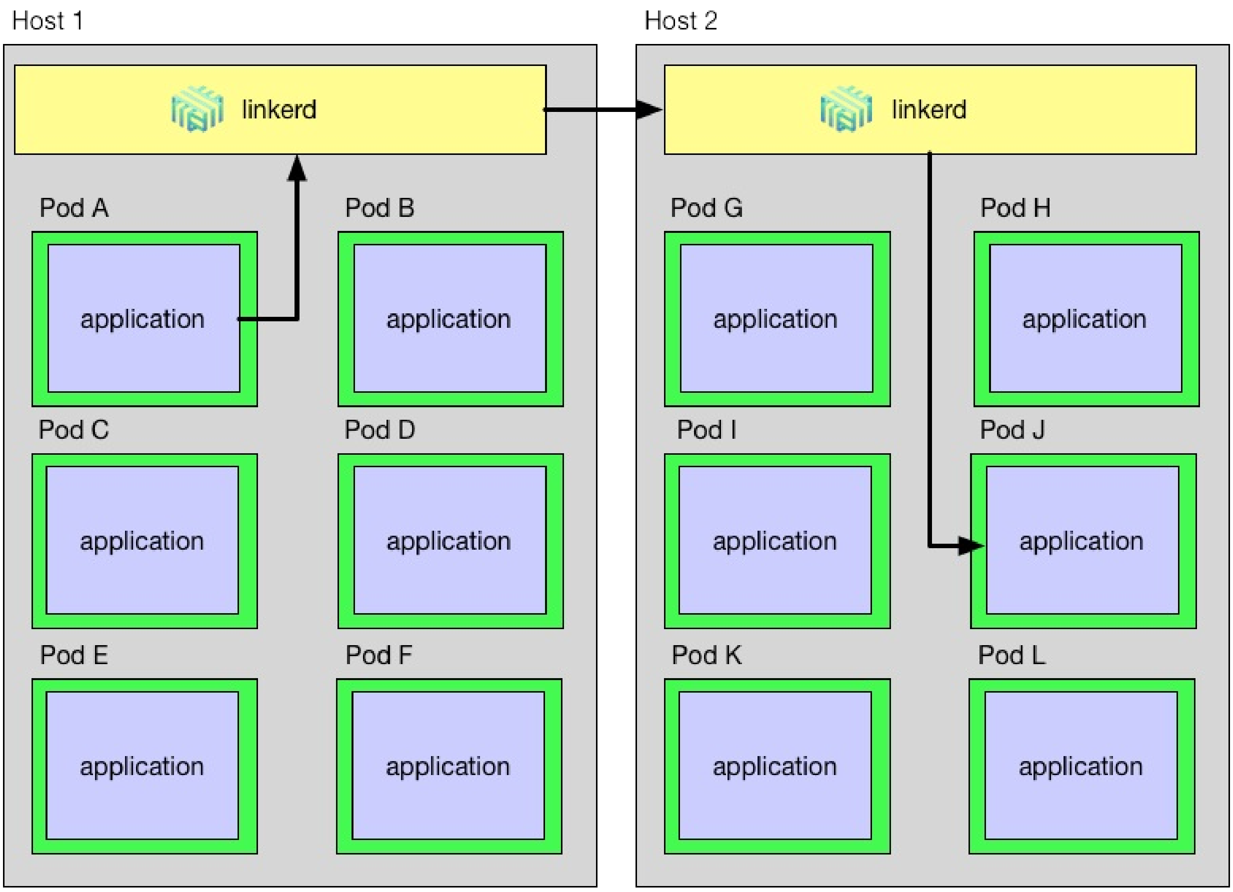
- 生产拓扑
- 服务发现机制
- 负载均衡和连接管理逻辑

应用程序也将从一致的全局流量控制系统中受益。这对于多语言应用程序尤其重要，因为通过库来实现这种一致性是非常困难的。

Linkerd实例可以作为sidecar（既为每个应用实体或每个主机部署一个实例）来运行。 由于linkerd实例是无状态和独立的，因此它们可以轻松适应现有的部署拓扑。它们可以与各种配置的应用程序代码一起部署，并且基本不需要去协调它们。

详解

Linkerd 的基于 Kubernetes 的 Service Mesh 部署方式是使用 Kubernetes 中的 **DaemonSet** 资源对象，如下图所示。



图片 - *Linkerd* 部署架构（图片来自*https://buoyant.io/2016/10/14/a-service-mesh-for-kubernetes-part-ii-pods-are-great-until-theyre-not/*）

这样 Kubernetes 集群中的每个节点都会运行一个 Linkerd 的 Pod。

但是这样做就会有几个问题：

- 节点上的应用如何发现其所在节点上的 Linkerd 呢？
- 节点间的 Linkerd 如何路由的呢？
- Linkerd 如何将接收到的流量路由到正确的目的应用呢？
- 如何对应用的路有做细粒度的控制？

这几个问题在 Buoyant 公司的这篇博客中都有解答：A Service Mesh for Kubernetes, Part II: Pods are great until they're not，我们下面将简要的回答上述问题。

节点上的应用如何发现其所在节点上的 Linkerd 呢？

简而言之，是使用环境变量的方式，如在应用程序中注入环境变量 `http_proxy` ：

```
env:
- name: NODE_NAME
  valueFrom:
    fieldRef:
      fieldPath: spec.nodeName
- name: http_proxy
  value: $(NODE_NAME):4140
args:
- "-addr=:7777"
- "-text=Hello"
- "-target=world"
```

这要求应用程序必须支持该环境变量，为应用程序所在的 Pod 设置了一个代理，实际上对于每种不同的协议 Linkerd 都监听不同的端口。

- 4140 for HTTP
- 4240 for HTTP/2
- 4340 for gRPC

关于 Linkerd 作为 Service Mesh 的详细配置请参考 serivcemesh.yml。

节点间的 Linkerd 如何路由的以及 Linkerd 如何将接收到的流量路由到正确的目的应用呢？

通过 **transformer** 来确定节点间的 Linkerd 路由，参考下面的配置：

```
routers:
- protocol: http
  label: outgoing
  interpreter:
    kind: default
    transformers:
    - kind: io.l5d.k8s.daemonset
      namespace: default
      port: incoming
      service: l5d
```

Router 定义 Linkerd 如何实际地处理流量。Router 监听请求并在这些请求上应用路有规则，代理这些请求到正确的目的地。Router 是与协议相关的。对于每个 Router 都需要定义一个 incoming router 和一个 outcoming router。预计该应用程

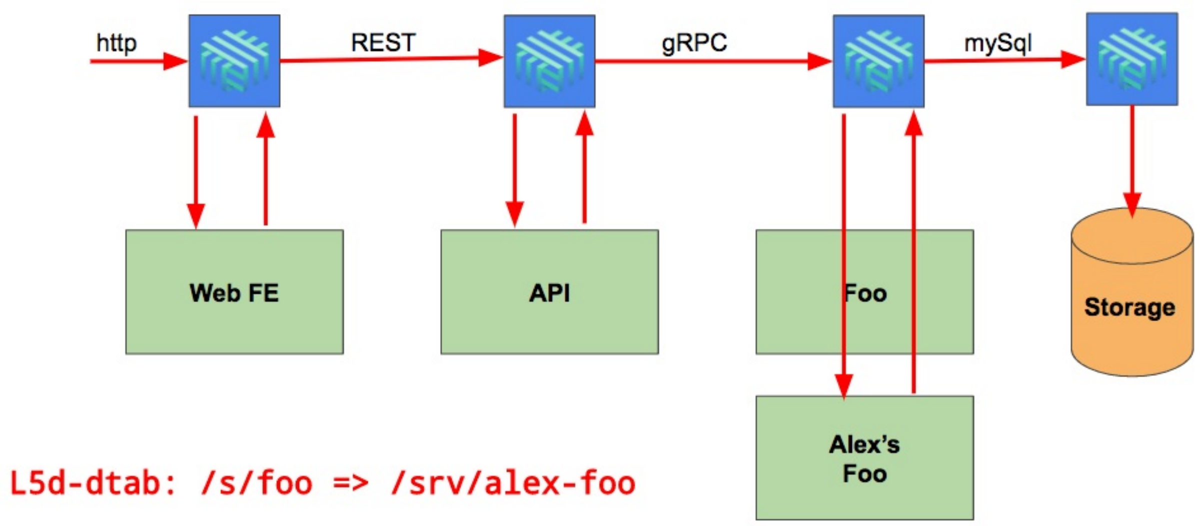
序将流量发送到 outcoming router，该 outcoming router 将其代理到目标服务节点上运行的 Linkerd 的 incoming router。Incoming router 后将请求代理给目标应用程序本身。我们还定义了 HTTP 和 HTTP/2 incoming router，它们充当 Ingress controller 并基于 Ingress 资源进行路由。

如何对路由规则做细粒度的控制呢？

路由规则配置是在 namerd 中进行的，例如：

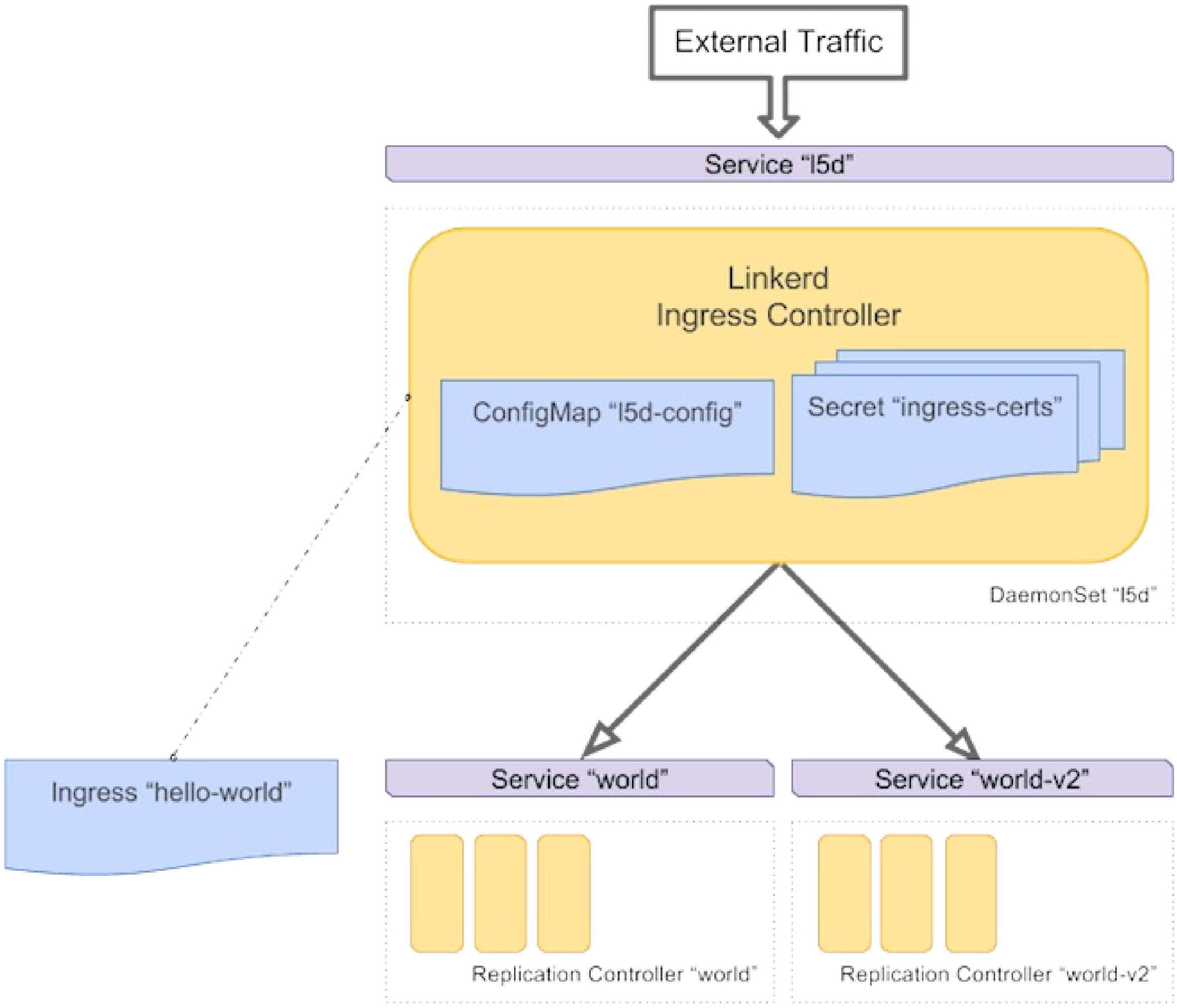
```
$ namerctl dtab get internal
# version MjgzNjk5NzI=
/srv          => /#/io.l5d.k8s/default/http ;
/host         => /srv ;
/tmp          => /srv ;
/svc          => /host ;
/host/world   => /srv/world-v1 ;
```

Namerd 中存储了很多 dtab 配置，通过这些配置来管理路有规则，实现微服务的流量管理。



图片 - 基于 *dtab* 的路由规则配置阶段发布

如果将 Linkerd 作为边缘节点还可以充当 Ingress controller，如下图所示。



图片 - *Linkerd ingress controller*

Linkerd 自己最令人称道的是它在每台主机上只安装一个 Pod，如果使用 Sidecar 模式会为每个应用程序示例旁都运行一个容器，这样会导致过多的资源消耗。Squeezing blood from a stone: small-memory JVM techniques for microservice sidecars 这篇文章中详细说明了 Linkerd 的资源消耗与性能。

参考

- A Service Mesh for Kubernetes, Part II: Pods are great until they're not
- Squeezing blood from a stone: small-memory JVM techniques for microservice sidecars
- Buoyant发布服务网格Linkerd的1.0版本

- Linkerd documentation
- Istio：用于微服务的服务啮合层

Updated at 2018-05-09 14:15:47

Linkerd 使用指南

本文是 Linkerd 使用指南，我们着重讲解在 kubernetes 中如何使用 linkerd 作为 kubernetes 的 Ingress controller。

前言

Linkerd 作为一款 service mesh 与kubernetes 结合后主要有以下几种用法：

1. 作为服务网关，可以监控 kubernetes 中的服务和实例
2. 使用 TLS 加密服务
3. 通过流量转移到持续交付
4. 开发测试环境（Eat your own dog food）、Ingress 和边缘路由
5. 给微服务做 staging
6. 分布式 tracing
7. 作为 Ingress controller
8. 使用 gRPC 更方便

以下我们着重讲解在 kubernetes 中如何使用 linkerd 作为 kubernetes 的 Ingress controller，并作为边缘节点代替 Traefik 的功能，详见 边缘节点的配置。

准备

安装测试时需要用到的镜像有：

```
buoyantio/helloworld:0.1.4
buoyantio/jenkins-plus:2.60.1
buoyantio/kubectl:v1.4.0
buoyantio/linkerd:1.1.2
buoyantio/namerd:1.1.2
buoyantio/nginx:1.10.2
linkerd/namerctl:0.8.6
openzipkin/zipkin:1.20
tutum/dnsutils:latest
```

这些镜像可以直接通过 Docker Hub 获取，我将它们下载下来并上传到了自己的私有镜像仓库 `harbor-001.jimmysong.io` 中，下文中用到的镜像皆来自我的私有镜像仓库，yaml 配置见 linkerd 目录，并在使用时将配置中的镜像地址修改为你自己的。

部署

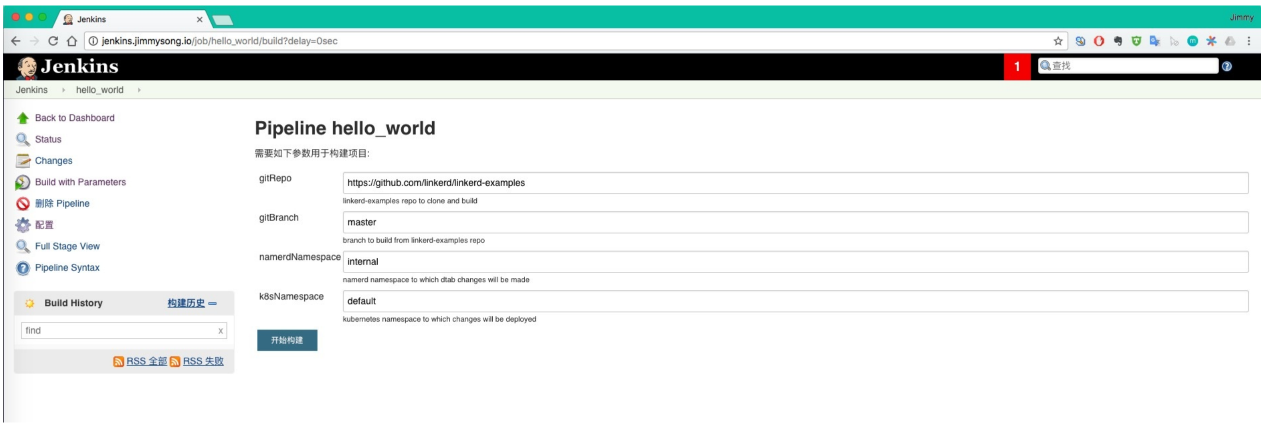
首先需要先创建 RBAC，因为使用 namerd 和 ingress 时需要用到。

```
$ kubectl create -f linkerd-rbac-beta.yml
```

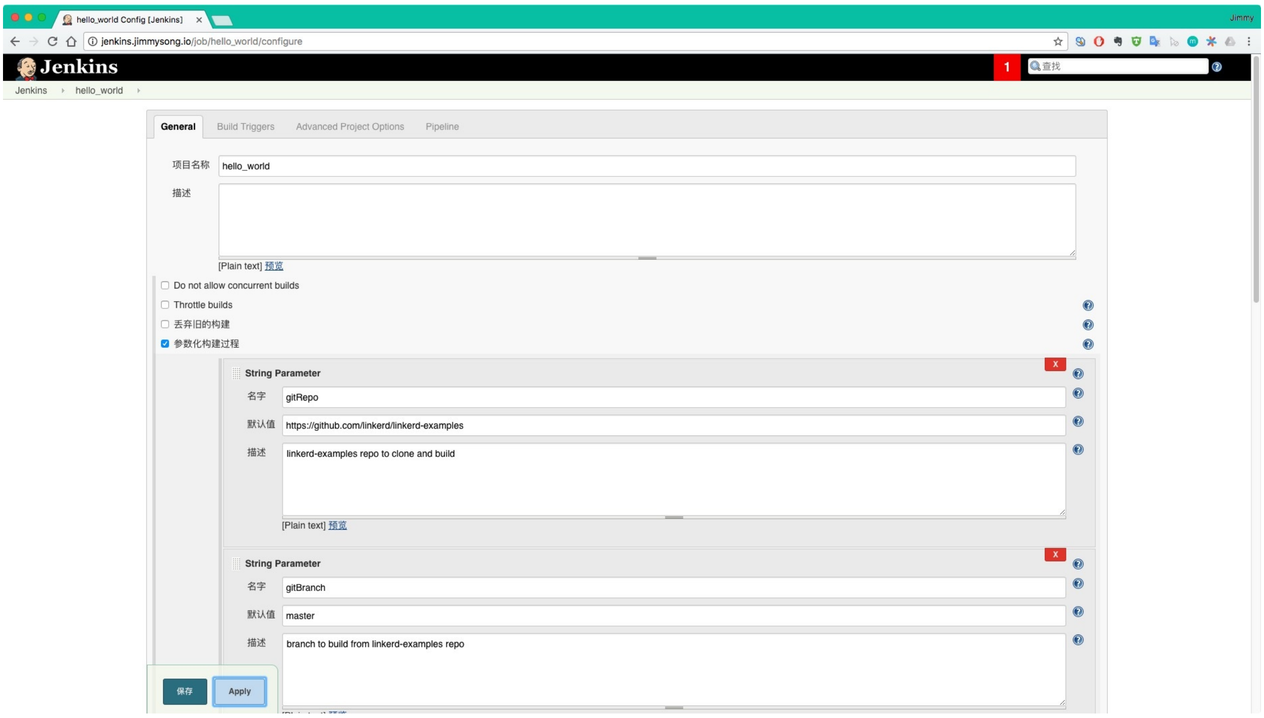
Linkerd 提供了 Jenkins 示例，在部署的时候使用以下命令：

```
$ kubectl create -f jenkins-rbac-beta.yml
$ kubectl create -f jenkins.yml
```

访问 http://jenkins.jimmysong.io



图片 - *Jenkins pipeline*



图片 - *Jenkins config*

注意：要访问 Jenkins 需要在 Ingress 中增加配置，下文会提到。

在 kubernetes 中使用 Jenkins 的时候需要注意 Pipeline 中的配置：

```
    def currentVersion = getCurrentVersion()
    def newVersion = getNextVersion(currentVersion)
    def frontendIp = kubectl("get svc l5d -o jsonpath=\"{.status
.loadBalancer.ingress[0].*}\"").trim()
    def originalDst = getDst(getDtab())
```

`frontendIP` 的地址要配置成 service 的 Cluster IP ，因为我们没有用到 LoadBalancer。

需要安装 namerd，namerd 负责 dtab 信息的存储，当然也可以存储在 etcd、consul中。dtab 保存的是路由规则信息，支持递归解析，详见 dtab。

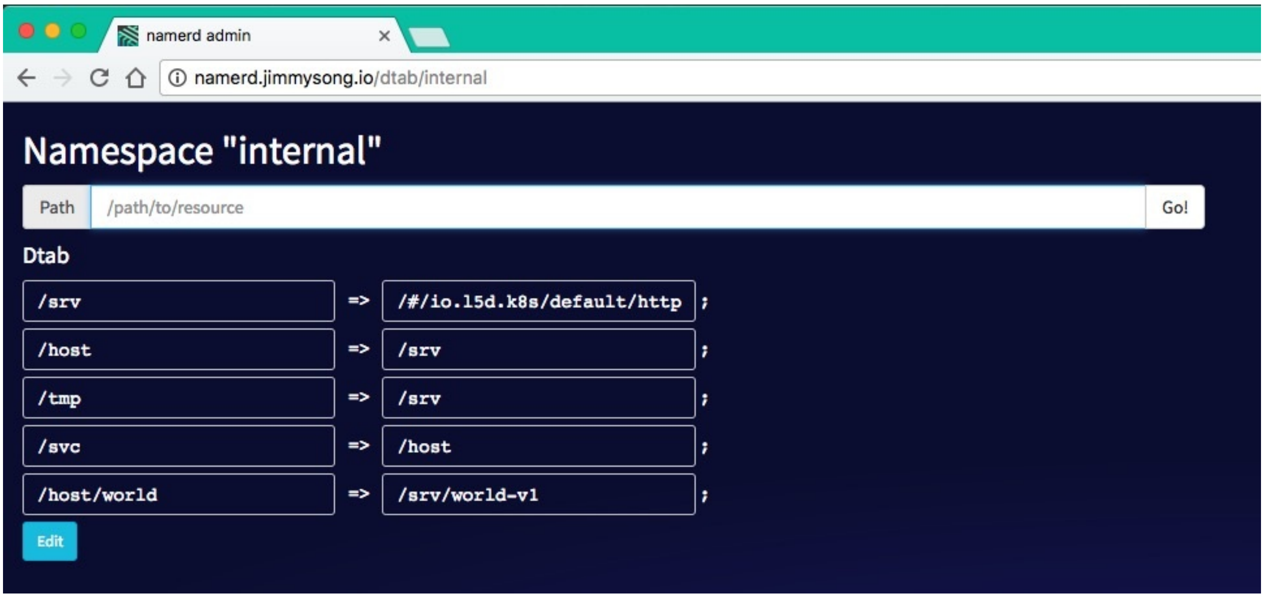
流量切换主要是通过 dtab 来实现的，通过在 HTTP 请求的 header 中增加 `l5d-dtab` 和 `Host` 信息可以对流量分离到 kubernetes 中的不同 service 上。

遇到的问题

Failed with the following error(s) Error signal dtab is already marked as being deployed!

因为该 dtab entry 已经存在，需要删除后再运行。

访问 http://namerd.jimmysong.io



图片 - *namerd*

dtab 保存在 namerd 中，该页面中的更改不会生效，需要使用命令行来操作。

使用 namerctl 来操作。

```
$ namerctl --base-url http://namerd-backend.jimmysong.io dtab up
date internal file
```

注意：update 时需要将更新文本先写入文件中。

部署 Linkerd

直接使用 yaml 文件部署，注意修改镜像仓库地址。

```
# 创建 namerd
$ kubectl create -f namerd.yaml
# 创建 ingress
$ kubectl create -f linkerd-ingress.yml
# 创建测试服务 hello-world
$ kubectl create -f hello-world.yml
# 创建 API 服务
$ kubectl create -f api.yml
# 创建测试服务 world-v2
$ kubectl create -f world-v2.yml
```

为了在本地调试 linkerd，我们将 linkerd 的 service 加入到 ingress 中，详见 边缘节点配置。

在 Ingress 中增加如下内容：

```yaml
  - host: linkerd.jimmysong.io
    http:
      paths:
      - path: /
        backend:
          serviceName: l5d
          servicePort: 9990
  - host: linkerd-viz.jimmysong.io
    http:
      paths:
      - path: /
        backend:
          serviceName: linkerd-viz
          servicePort: 80
  - host: l5d.jimmysong.io
    http:
      paths:
      - path: /
        backend:
          serviceName: l5d
          servicePort: 4141
  - host: jenkins.jimmysong.io
    http:
      paths:
      - path: /
        backend:
          serviceName: jenkins
          servicePort: 80
```

在本地 `/etc/hosts` 中添加如下内容：

```
172.20.0.119 linkerd.jimmysong.io
172.20.0.119 linkerd-viz.jimmysong.io
172.20.0.119 l5d.jimmysong.io
```

测试路由功能

使用 curl 简单测试。

单条测试

```
$ curl -s -H "Host: www.hello.world" 172.20.0.120:4141
Hello (172.30.60.14) world (172.30.71.19)!!%
```

请注意请求返回的结果，表示访问的是 `world-v1` service。

```
$ for i in $(seq 0 10000);do echo $i;curl -s -H "Host: www.hello
.world" 172.20.0.120:4141;done
```

使用 ab test。

```
$ ab -c 4 -n 10000  -H "Host: www.hello.world" http://172.20.0.1
20:4141/
This is ApacheBench, Version 2.3 <$Revision: 1757674 $>
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeust
ech.net/
Licensed to The Apache Software Foundation, http://www.apache.or
g/

Benchmarking 172.20.0.120 (be patient)
Completed 1000 requests
Completed 2000 requests
Completed 3000 requests
Completed 4000 requests
Completed 5000 requests
Completed 6000 requests
Completed 7000 requests
Completed 8000 requests
Completed 9000 requests
Completed 10000 requests
Finished 10000 requests


Server Software:
Server Hostname:        172.20.0.120
Server Port:            4141
```

```
Document Path:          /
Document Length:        43 bytes

Concurrency Level:      4
Time taken for tests:   262.505 seconds
Complete requests:      10000
Failed requests:        0
Total transferred:      2210000 bytes
HTML transferred:       430000 bytes
Requests per second:    38.09 [#/sec] (mean)
Time per request:       105.002 [ms] (mean)
Time per request:       26.250 [ms] (mean, across all concurrent
 requests)
Transfer rate:          8.22 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:       36   51  91.1     39    2122
Processing:    39   54  29.3     46     585
Waiting:       39   52  20.3     46     362
Total:         76  105  96.3     88    2216

Percentage of the requests served within a certain time (ms)
  50%     88
  66%     93
  75%     99
  80%    103
  90%    119
  95%    146
  98%    253
  99%    397
 100%   2216 (longest request)
```

监控 kubernets 中的服务与实例

访问 http://linkerd.jimmysong.io 查看流量情况

Outcoming



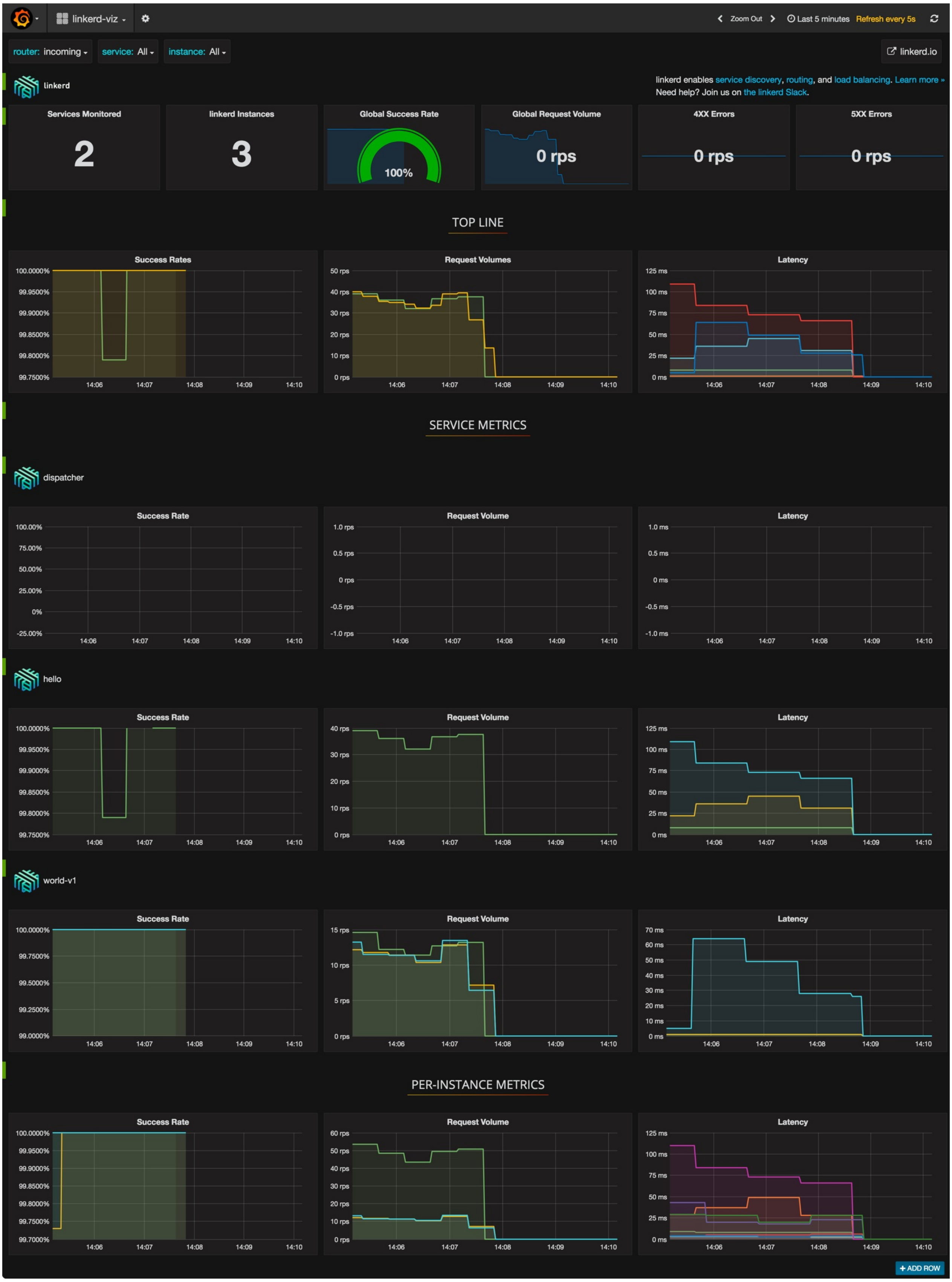
图片 - *linkerd*监控

Incoming



图片 - *linkerd*监控

访问 http://linkerd-viz.jimmysong.io 查看应用 metric 监控



图片 - *linkerd*性能监控

测试路由

测试在 http header 中增加 dtab 规则。

```
$ curl -H "Host: www.hello.world" -H "l5d-dtab:/host/world => /s
rv/world-v2;" 172.20.0.120:4141
Hello (172.30.60.14) earth (172.30.94.40)!!
```

请注意调用返回的结果，表示调用的是 `world-v2` 的 service。

另外再对比 ab test 的结果与 `linkerd-viz` 页面上的结果，可以看到结果一致。

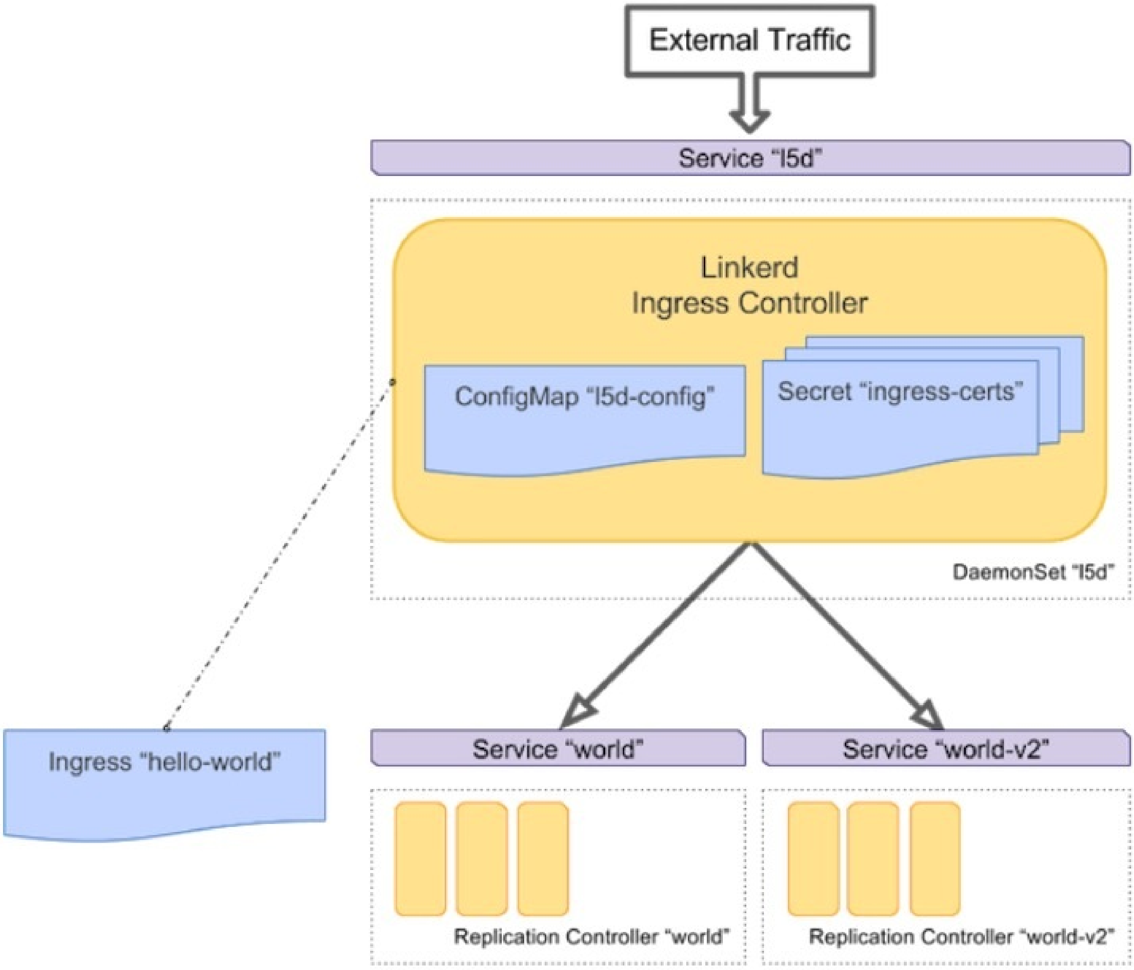
但是我们可能不想把该功能暴露给所有人，所以可以在前端部署一个 nginx 来过滤 header 中的 `l5d-dtab` 打头的字段，并通过设置 cookie 的方式来替代 header 里的 `l5d-dtab` 字段。

```
$ http_proxy=http://172.20.0.120:4141 curl -s http:/hello
Hello (172.30.60.14) world (172.30.71.19)!!
```

将 Linkerd 作为 Ingress controller

将 Linkerd 作为 kubernetes ingress controller 的方式跟将 Treafik 作为 ingress controller 的过程过程完全一样，可以直接参考 边缘节点配置。

架构如下图所示。



图片 - *Linkerd ingress controller*

(图片来自 A Service Mesh for Kubernetes - Buoyant.io)

当然可以绕过 kubernetes ingress controller 直接使用 linkerd 作为边界路由，通过 dtab 和 linkerd 前面的 nginx 来路由流量。

参考

- https://github.com/linkerd/linkerd-examples
- [A Service Mesh for Kubernetes]
- [dtab]

Conduit - 基于Kubernetes的轻量级Service Mesh

2017年12月在得克萨斯州的Asdin，KubeCon和CloudNativeCon上，创造了Service Mesh这个词汇并开源了Linkerd的公司Buoyant，又开源了一款针对Kubernetes的超轻量Service Sesh——Conduit。它可以透明得管理服务运行时之间的通信，使得在Kubernetes上运行服务更加安全和可靠；它还具有不用修改任何应用程序代码即可改进应用程序的可观测性、可靠性及安全性等方面的特性。

Condiut与Linkerd的设计方式不同，它跟Istio一样使用的是Sidecar模式，但架构又没Istio那么复杂。Conduit只支持Kubernetes，且只支持HTTP2（包括gRPC）协议。

Conduit使用Rust和Go语言开发，GitHub地址https://github.com/runconduit/conduit

安装Conduit必须使用Kubernetes1.8以上版本。

参考

Conduit GitHub：https://github.com/runconduit/conduit

关于Conduit的更多资源请参考官方网站：https://conduit.io/

另外，我们翻译Conduit的官方文档见：https://github.com/doczhcn/conduit

关于Service Mesh的更多内容请访问Service Mesh中国：http://www.servicemesh.cn/

Conduit 概览

Conduit是一款针对Kubernetes的超轻量级的service mesh。它可以透明得管理服务运行时之间的通信，使得在Kubernetes上运行服务更加安全和可靠；它还具有不用修改任何应用程序代码即可改进应用程序的可观测性、可靠性及安全性等方面的特性。

本文档将从一个较高层次介绍Conduit及其是如何工作的。如果不熟悉service mesh模型，或许可以先阅读William Morgan的概览文章 什么是service mesh？为什么需要它？

Conduit架构

Conduit service mesh部署到Kubernetes集群时有两个基本组件：数据平面和控制平面。数据平面承载服务实例间实际的应用请求流量，而控制平面则驱动数据平面并修改其行为（及访问聚合指标）提供API。Conduit CLI和Web UI使用此API并为人类提供符合人体工程学的控制。

让我们依次认识这些组件。

Conduit的数据平面由轻量级的代理组成，这些代理作为sidecar容器与每个服务代码的实例部署在一起。如果将服务“添加”到Conduit servie mesh中，必须重新部署该服务的pod，以便在每个pod中包含一个数据平面代理。（ `conduit inject` 命令可以完成这个任务，以及透明地从每个实例通过代理汇集流量所需的配置工作。）

这些代理透明地拦截进出每个pod的通信，并增加诸如重试和超时、检测及加密（TLS）等特性，并根据相关策略来允许和拒绝请求。

这些代理并未设计成通过手动方式配置；相反，它们的行为是由控制平面驱动的。

Conduit控制平面是一组运行在专用Kubernetes名称空间（默认情况下为 `conduit` ）的服务。这些服务完成各种事情 - 聚合遥测数据，提供面向用户的API，向数据平面代理提供控制数据等。它们一起驱动数据平面的行为。

使用Conduit

为了支持Conduit的人机交互，可以使用Conduit CLI及web UI（也可以通过相关工具比如 `kubectl` ）。CLI 和 web UI通过API驱动控制平面，而控制平面相应地驱动数据平面的行为。

控制平面API设计得足够通用，以便能基于此构建其他工具。比如，你可能希望从另外一个CI/CD系统来驱动API。

运行 `conduit --help` 可查看关于CLI功能的简短概述。

Conduit 与 Kubernetes

Conduit设计用于无缝地融入现有的Kubernetes系统。该设计有几个重要特征。

第一，Conduit CLI（ `conduit` ）设计成尽可能地与 `kubectl` 一起使用。比如， `conduit install` 和 `conduit inject` 生成的Kubernetes配置，被设计成直接送入 `kubectl` 。这是为了在service mesh和编排系统之间提供一个明确的分工，并且使得Conduit适配现有的Kubernetes工作流程。

第二，Kubernetes中Conduit的核心词是Deployment，而不是Service。举个例子， `conduit inject` 增加一个Deployment，Conduit web UI会显示这些Deployment，并且每个Deployment都会给出聚合的性能指标。这是因为单个pod可以是任意数量Service的一部分，而这会导致流量与pod之间出现复杂的映射。相比之下，Deployment要求单个pod只能属于一个Deployment的一部分。通过基于Deployment而不是Service来构建，流量与pod间的映射就总是清晰的。

这两个设计特性能很好地组合。比如， `conduit inject` 可用于一个运行的Deployment，因为当它更新Deployment时， Kubernetes会回滚pod以包括数据平面代理。

扩展Conduit的行为

Conduit控制平面还为构建自定义功能提供了一个便捷的入口。Conduit最初发布时并不支持这一点，在不远的将来，通过编写gRPC插件来作为控制平面的一部分运行，将能扩展Conduit的功能，而无需重新编译Conduit。

安装Conduit

本文档指导您如何在kubernetes上安装Conduit service mesh。

前提条件

- kubernetes版本为1.8或以上

用到的镜像如下：

- buoyantio/kubectl:v1.6.2
- gcr.io/runconduit/controller:v0.1.0
- gcr.io/runconduit/web:v0.1.0
- prom/prometheus:v1.8.1

其中位于gcr.io的镜像我备份到了DockerHub：

- jimmysong/runconduit-web:v0.1.0
- jimmysong/runconduit-controller:v0.1.0

另外两个镜像本身就可以从DockerHub上下载。

部署

到release页面上下载conduit的二进制文件。

使用 `conduit install` 命令生成了用于部署到kubernetes中yaml文件，然后修改文件中的镜像仓库地址为你自己的镜像地址。

```
conduit install>conduit-0.1.0.yaml
# 修改完镜像地址执行
kubectl apply -f conduit-0.1.0.yaml
```

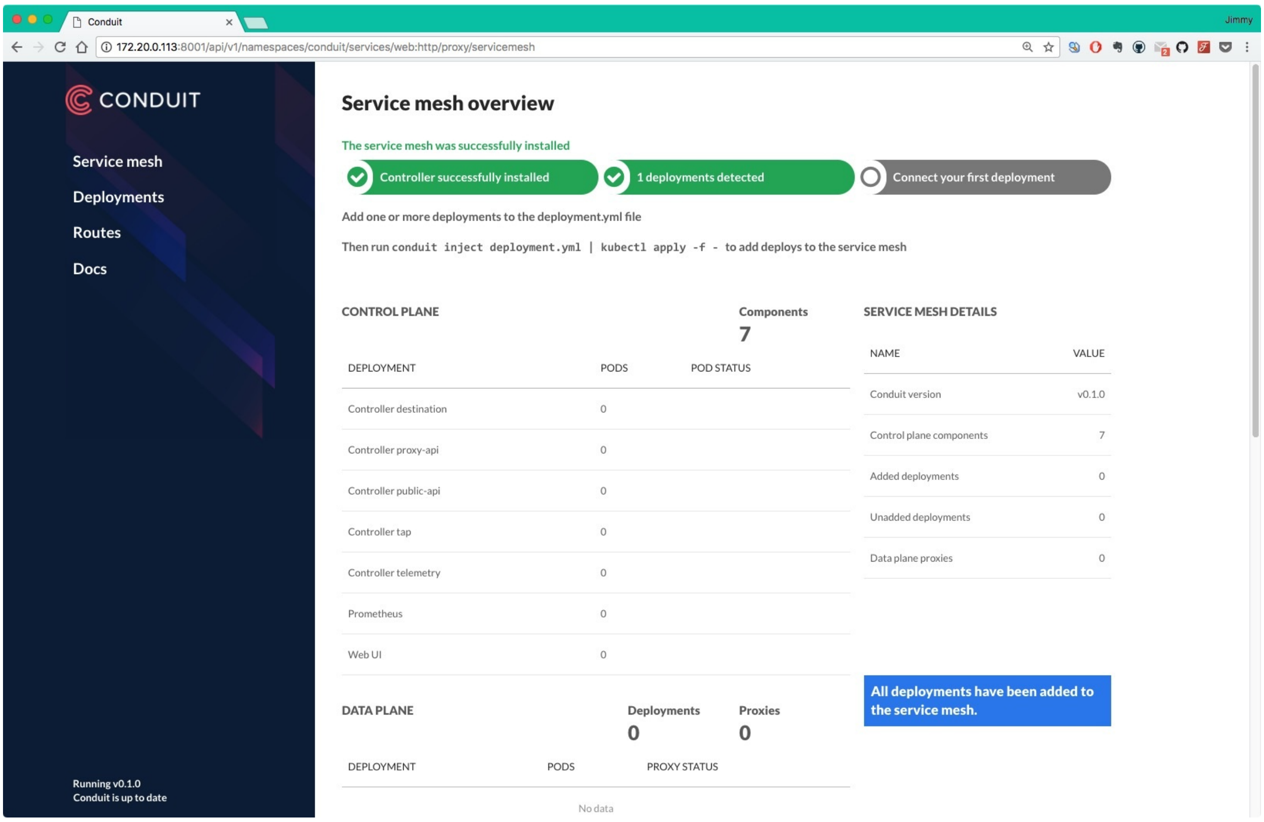
修改后的yaml文件见：conduit-0.1.0.yaml。

注意：Conduit官方给出的yaml文件中不包括RBAC授权，我重新修改了，增加了RBAC和ServiceAccount。

使用 `kubectl proxy` 来开放外网访问conduit dashboard：

```
kubectl proxy --address='172.20.0.113' --port=8001 --accept-hosts
='^*$'
```

在浏览器中访问 http://172.20.0.113:8001/api/v1/namespaces/conduit/services/web:http/proxy/servicemesh将看到如下页面：



图片 - *Conduit dashboard*

Conduit inject

Conduit注入的时候需要用到如下两个镜像：

- gcr.io/runconduit/proxy:v0.1.0
- gcr.io/runconduit/proxy-init:v0.1.0

我将其备份到了DockerHub：

- jimmysong/runconduit-proxy:v0.1.0
- jimmysong/runconduit-proxy-init:v0.1.0

查看conduit向yaml文件中注入了哪些配置，我们使用my-nginx.yaml为例：

```
conduit inject --init-image harbor-001.jimmysong.io/library/runc
onduit-proxy-init --proxy-image harbor-001.jimmysong.io/library/
runconduit-proxy my-nginx.yaml|kubectl apply -f -
```

注意：只需要指定镜像名称即可，tag与使用的conduit server版本相同，会自动注入。

my-nginx.yaml的内容如下：

```yaml
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: my-nginx
spec:
  replicas: 2
  template:
    metadata:
      labels:
        run: my-nginx
    spec:
      containers:
      - name: my-nginx
        image: harbor-001.jimmysong.io/library/nginx:1.9
        ports:
        - containerPort: 80
---
apiVersion: v1
kind: Service
metadata:
  name: my-nginx
  labels:
    app: my-nginx
spec:
  ports:
  - port: 80
    protocol: TCP
    name: http
  selector:
    run: my-nginx
```

Conduit自动注入后生成的新的yaml文件内容如下：

```yaml
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  creationTimestamp: null
  name: my-nginx
spec:
  replicas: 2
  strategy: {}
  template:
    metadata:
      annotations:
        conduit.io/created-by: conduit/cli v0.1.0
        conduit.io/proxy-version: v0.1.0
      creationTimestamp: null
      labels:
        conduit.io/controller: conduit
        conduit.io/plane: data
        run: my-nginx
    spec:
      containers:
      - image: harbor-001.jimmysong.io/library/nginx:1.9
        name: my-nginx
        ports:
        - containerPort: 80
        resources: {}
      - env:
        - name: CONDUIT_PROXY_LOG
          value: trace,h2=debug,mio=info,tokio_core=info
        - name: CONDUIT_PROXY_CONTROL_URL
          value: tcp://proxy-api.conduit.svc.cluster.local:8086
        - name: CONDUIT_PROXY_CONTROL_LISTENER
          value: tcp://0.0.0.0:4190
        - name: CONDUIT_PROXY_PRIVATE_LISTENER
          value: tcp://127.0.0.1:4140
        - name: CONDUIT_PROXY_PUBLIC_LISTENER
          value: tcp://0.0.0.0:4143
        - name: CONDUIT_PROXY_NODE_NAME
          valueFrom:
            fieldRef:
              fieldPath: spec.nodeName
        - name: CONDUIT_PROXY_POD_NAME
          valueFrom:
            fieldRef:
              fieldPath: metadata.name
        - name: CONDUIT_PROXY_POD_NAMESPACE
          valueFrom:
            fieldRef:
              fieldPath: metadata.namespace
        image: harbor-001.jimmysong.io/library/runconduit-proxy:
v0.1.0
        imagePullPolicy: IfNotPresent
        name: conduit-proxy
```

```
        ports:
        - containerPort: 4143
          name: conduit-proxy
        resources: {}
        securityContext:
          runAsUser: 2102
      initContainers:
      - args:
        - -p
        - "4143"
        - -o
        - "4140"
        - -i
        - "4190"
        - -u
        - "2102"
        image: harbor-001.jimmysong.io/library/runconduit-proxy-
init:v0.1.0
        imagePullPolicy: IfNotPresent
        name: conduit-init
        resources: {}
        securityContext:
          capabilities:
            add:
            - NET_ADMIN
          privileged: false
status: {}
---
apiVersion: v1
kind: Service
metadata:
  name: my-nginx
  labels:
    app: my-nginx
spec:
  ports:
  - port: 80
    protocol: TCP
    name: http
  selector:
    run: my-nginx
---
```

部署示例应用

使用下面的命令部署官方提供的示例应用：

```
curl https://raw.githubusercontent.com/rootsongjc/kubernetes-han
dbook/master/manifests/conduit/emojivoto.yml | conduit inject --
init-image harbor-001.jimmysong.io/library/runconduit-proxy-init
 --proxy-image harbor-001.jimmysong.io/library/runconduit-proxy
- --skip-inbound-ports=80 | kubectl apply -f -
```

注意：其中使用的镜像地址已经改为我的私有镜像仓库地址，大家使用时请注意修改。

参考

- Getting started - conduit.io

Envoy

Envoy 是一款由 Lyft 开源的，使用 C++ 编写的 L7 代理和通信总线，目前是 CNCF 旗下的开源项目，代码托管在 GitHub 上，它也是 Istio service mesh 中默认的 data plane。

特性

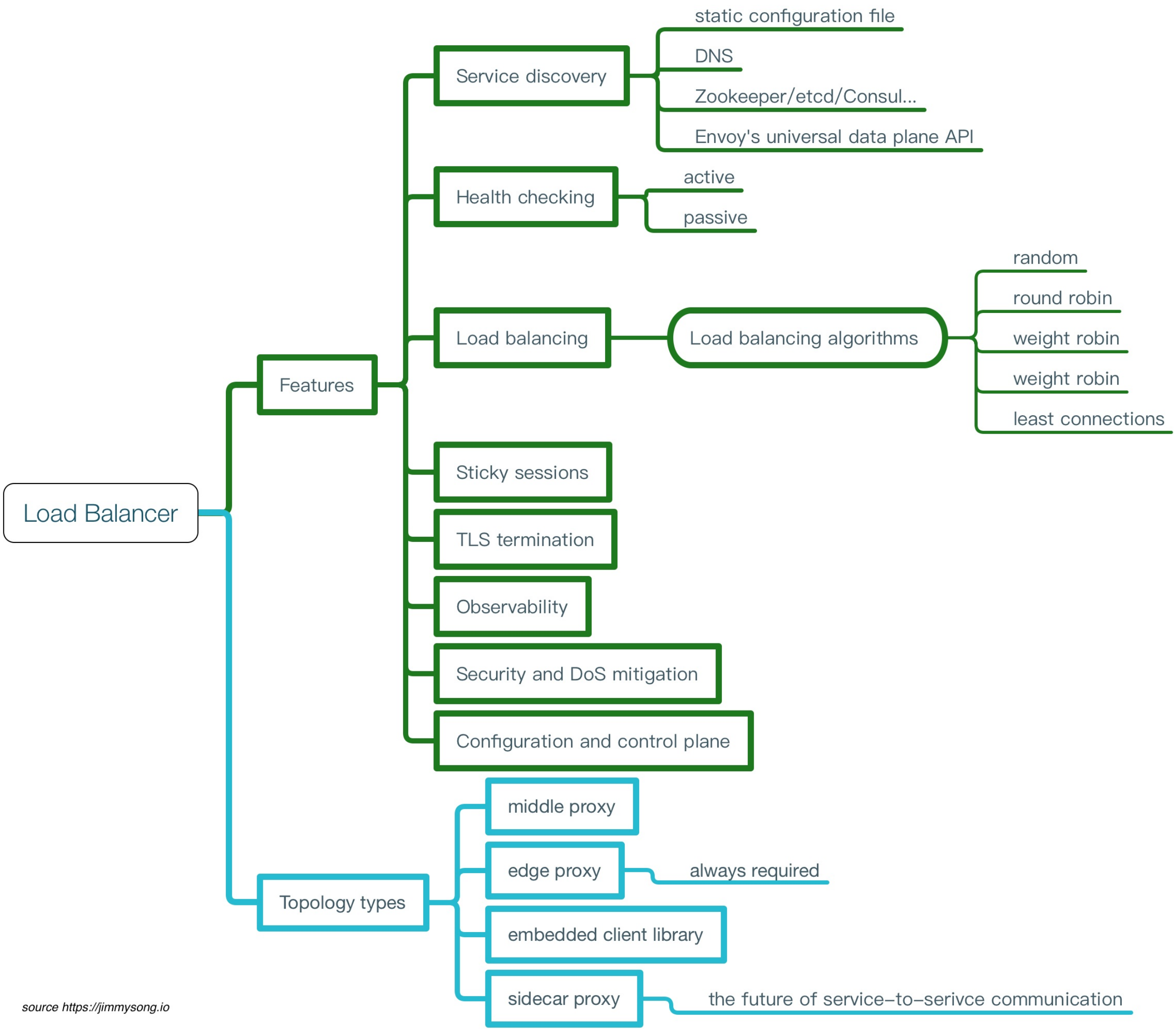
Envoy 包括如下特性：

- 进程外架构，不侵入应用进程
- 使用现代版 C++11 代码
- L3/L4 filter 架构
- HTTP L7 filter 架构
- 支持 HTTP/2
- HTTP L7 routing
- 支持 gRPC
- 支持 MongoDB L7
- 动态配置
- 最佳可观测性
- 支持 front/edge proxy
- 高级负载均衡
- 健康检查
- 服务发现
- 支持 DynamoDB L7

Envoy 本身无法构成一个完整的 Service Mesh，但是它可以作为 service mesh 中的应用间流量的代理，负责 service mesh 中的数据层。

负载均衡与代理

在 Matt Klein（Envoy 的作者） 的 Introduction to modern network load balancing and proxying 文章详细描述了现代的网络负载均衡与代理的特性与模式，L4 与 L7 负载均衡的区别以及 L7 负载均衡的现状，总结如下图。



图片 - 负载均衡器的特性以及拓扑类型

Matt Klein 是在他的文章中指出 sidecar 模式的 proxy 将取代另外三种模式而成为服务间通信的负载均衡器。

参考

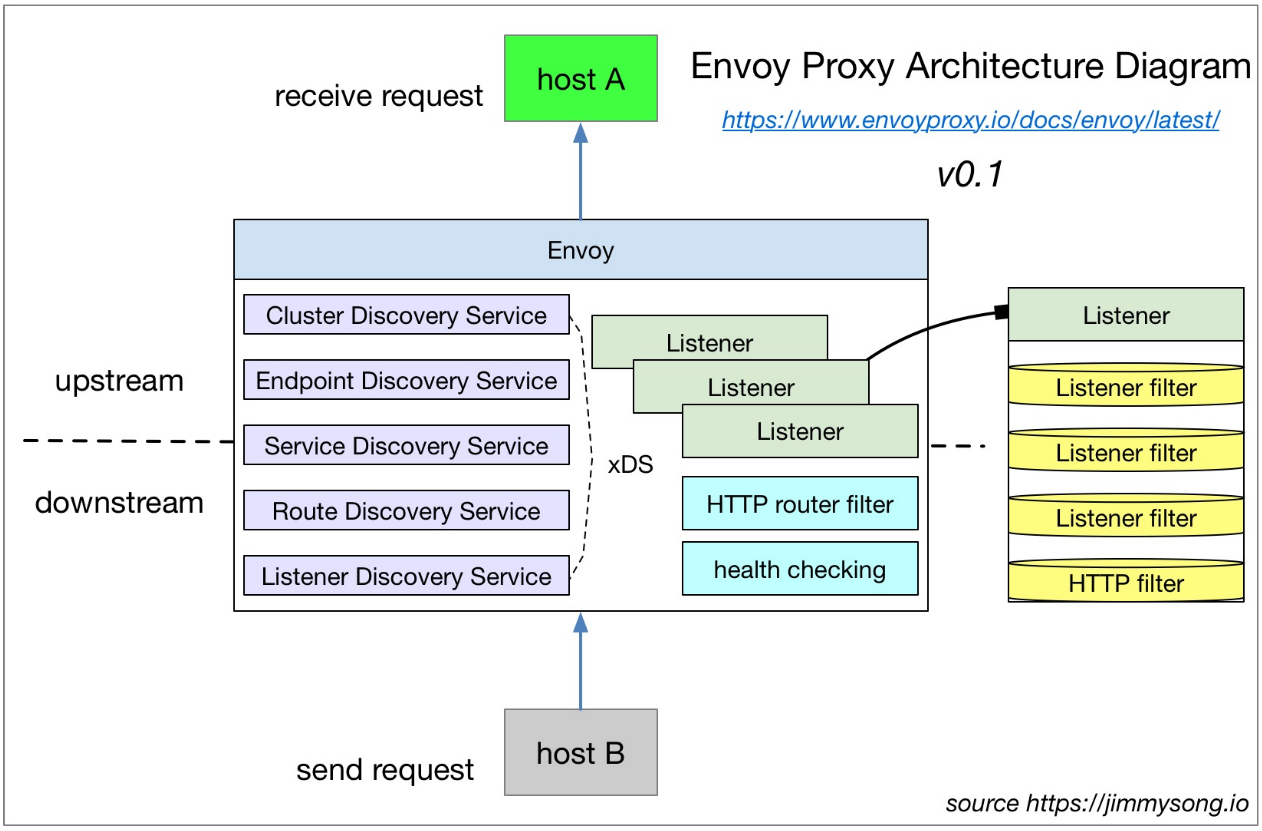
- Introduction to modern network load balancing and proxying
- 更多信息请参考 Envoy 官网。

Envoy 的架构与基本术语

在了解一门技术之前一开始就要了解其中的基本概念和术语，只有融入了该语境才能理解这门技术。本文将为大家介绍 Envoy 中的基本术语和重点概念。

架构

下图是 Envoy proxy 的架构图，显示了 host B 经过 Envoy 访问 host A 的过程。每个 host 上都可能运行多个 service，Envoy 中也可能有多个 Listener，每个 Listener 中可能会有多个 filter 组成了 chain。



图片 - *Envoy proxy* 架构图

其中的基本术语将在下面解释。

基本术语

Host：能够进行网络通信的实体（在手机或服务器等上的应用程序）。在 Envoy 中主机是指逻辑网络应用程序。只要每台主机都可以独立寻址，一块物理硬件上就运行多个主机。

Downstream：下游（downstream）主机连接到 Envoy，发送请求并或获得响应。

Upstream：上游（upstream）主机获取来自 Envoy 的链接请求和响应。

Cluster: 集群（cluster）是 Envoy 连接到的一组逻辑上相似的上游主机。Envoy 通过服务发现发现集群中的成员。Envoy 可以通过主动运行状况检查来确定集群成员的健康状况。Envoy 如何将请求路由到集群成员由负载均衡策略确定。

Mesh：一组互相协调以提供一致网络拓扑的主机。Envoy mesh 是指一组 Envoy 代理，它们构成了由多种不同服务和应用程序平台组成的分布式系统的消息传递基础。

运行时配置：与 Envoy 一起部署的带外实时配置系统。可以在无需重启 Envoy 或更改 Envoy 主配置的情况下，通过更改设置来影响操作。

Listener: 侦听器（listener）是可以由下游客户端连接的命名网络位置（例如，端口、unix域套接字等）。Envoy 公开一个或多个下游主机连接的侦听器。一般是每台主机运行一个 Envoy，使用单进程运行，但是每个进程中可以启动任意数量的 Listener（监听器），目前只监听 TCP，每个监听器都独立配置一定数量的（L3/L4）网络过滤器。Listenter 也可以通过 Listener Discovery Service（**LDS**）动态获取。

Listener filter：Listener 使用 listener filter（监听器过滤器）来操作链接的元数据。它的作用是在不更改 Envoy 的核心功能的情况下添加更多的集成功能。Listener filter 的 API 相对简单，因为这些过滤器最终是在新接受的套接字上运行。在链中可以互相衔接以支持更复杂的场景，例如调用速率限制。Envoy 已经包含了多个监听器过滤器。

Http Route Table：HTTP 的路由规则，例如请求的域名，Path 符合什么规则，转发给哪个 Cluster。

Health checking：健康检查会与SDS服务发现配合使用。但是，即使使用其他服务发现方式，也有相应需要进行主动健康检查的情况。详见 health checking。

xDS

xDS 是一个关键概念，它是一类发现服务的统称，其包括如下几类：

- CDS：Cluster Discovery Service
- EDS：Endpoint Discovery Service
- SDS：Service Discovery Service
- RDS：Route Discovery Service
- LDS：Listener Discovery Service

正是通过对 xDS 的请求来动态更新 Envoy 配置。

Envoy Mesh

Envoy Mesh 指的是由 envoy 做负载均衡和代理的 mesh。该 Mesh 中会包含两类 envoy：

- Edge envoy：即流量进出 mesh 时候的 envoy，相当于 kubernetes 中的 ingress。
- Service envoy：服务 envoy 是跟每个 serivce 实例一起运行的，应用程序无感知的进程外工具，在 kubernetes 中会与应用容器以 sidecar 形式运行在同一个 pod 中。

Envoy 即可以单独作为 edge envoy，也可以仅做 service envoy 使用，也可以两者同时使用。Mesh 中的所有 envoy 会共享路由信息。

Envoy 配置

Envoy 中的配置包括两大类：listenner 配置和 cluster 配置。

Listener 配置

我们知道 Envoy 中可以配置一组 listener 以实现复杂的处理逻辑。Listener 中设置监听的 TCP 端口，还有一组 filter 对这些端口上的数据流进行处理。如下所示，该示例来自使用Envoy 作为前端代理。

```yaml
listeners:
- address:
    socket_address:
      address: 0.0.0.0
      port_value: 80
  filter_chains:
  - filters:
    - name: envoy.http_connection_manager
      config:
        codec_type: auto
        stat_prefix: ingress_http
        route_config:
          name: local_route
          virtual_hosts:
          - name: backend
            domains:
            - "*"
            routes:
            - match:
                prefix: "/service/1"
              route:
                cluster: service1
            - match:
                prefix: "/service/2"
              route:
                cluster: service2
```

这是一个 `http_connection_manager` 例子，其中必须包含 `virtual_hosts` 配置，而 `virtual_hosts` 配置中必须包含以下几项配置：

- `name`：服务名称
- `domains`：DNS 域名，必须能跟 `virtual_host` 的 URL 匹配
- `routes`：路由列表

每个路由中还可以包含以下配置：

- `prefix`：URL 路径前缀
- `cluster`：处理该请求的 envoy cluster
- `timeout_ms`：当出错时的超时时间

如上面的例子中，我们还需要定义 `service1` cluster 和 `service2` cluster。

Cluster 配置

Cluster 是一组逻辑相似的主机配置，定义哪些主机属于一个服务，cluster 的配置中包含了服务发现和负载均衡方式配置。依然是参考使用Envoy 作为前端代理中的配置：

```yaml
clusters:
 - name: service1
   connect_timeout: 0.25s
   type: strict_dns
   lb_policy: round_robin
   http2_protocol_options: {}
   hosts:
   - socket_address:
       address: service1
       port_value: 80
 - name: service2
   connect_timeout: 0.25s
   type: strict_dns
   lb_policy: round_robin
   http2_protocol_options: {}
   hosts:
   - socket_address:
       address: service2
       port_value: 80
```

Cluster 的配置中至少包含以下信息：

- `name` ：cluster 名称，就是服务名称
- `type` ：该 cluster 怎么知道主机是否启动？即服务发现类型，有以下方式：
  - `static` ：监听 cluster 中的所有主机
  - `strict_dns` ：envoy 会监听 DNS，每个匹配的 A 记录都会认定为有效
  - `logical_dns` ：envoy 将使用 DNS 来增加主机，如果 DNS 不再返回该主机也不会删除这些主机信息
  - `sds` ：即 Serivce Discovery Serivce，envoy 访问外部的 REST 获取 cluster 成员信息
- `lb_type` ：cluster 的负载均衡类型，有以下方式：
  - `round_robin` ：轮询主机
  - `weighted_least_request` ：最近获得最少请求的主机
  - `random` ：随机
- `hosts` ：能够定义 cluster 中主机的 URL 地址，通常是 `tcp://` URL

参考

- Terminology - www.envoyproxy.io
- Part 1: Getting started with Envoy Proxy for microservices resilience
- Envoy作为前端代理

Envoy 作为前端代理

本文是使用 Envoy 作为前端代理的介绍，仅使用 docker 容器和 docker-compose 做编排在单机中运行，帮助我们从更底层了解 Envoy，当我们将 Envoy 作为 Istio Service Mesh 的 data panel 的时候将更加游刃有余。

快速开始

Envoy 中的所有规则配置跟 Kubernetes 一样都是通过 YAML 文件来完成的。在继续下面的步骤之前，首先克隆 Envoy 的 GitHub repo。

```
git clone https://github.com/envoyproxy/envoy.git
```

运行 sandbox 测试

Envoy 官方提供了以下打包用例：

- Front Proxy
- Zipkin Tracing
- Jaeger Tracing
- gRPC Bridge

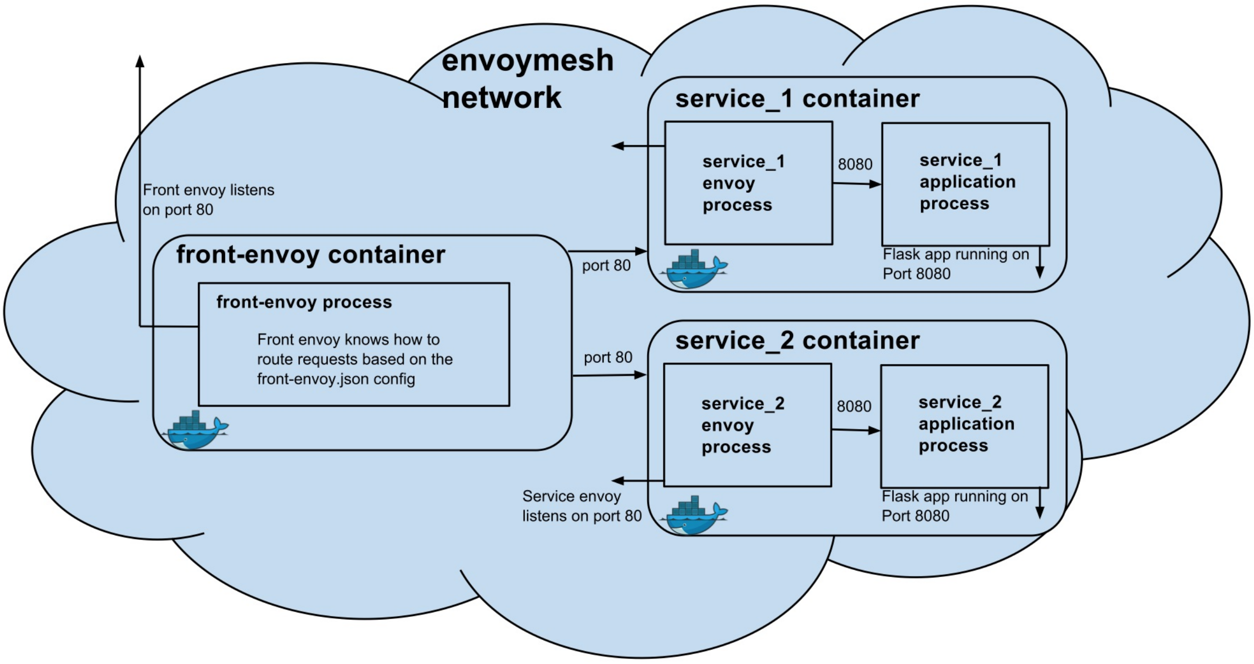
全部可以使用 `docker-compose` 运行，代码可以在 https://github.com/envoyproxy/envoy/tree/master/examples 找到。

Front proxy

Envoy 在 envoymesh 的边缘做反向代理，详细使用方式见 https://www.envoyproxy.io/docs/envoy/latest/start/sandboxes/front_proxy，在此我将解说下以下问题：

- Envoy 是如何作为进程外架构运行的？
- 为何说 Envoy 是无侵入式架构？
- Envoy 作为边缘反向代理能做什么？

本示例的架构图如下所示，此时 Envoy 将作为一个反向代理，类似于 Nginx，但与 Nginx 不同的是它还会作为一个进程，伴随每个服务一起运行在同一个容器中（在 Kubernetes 中可以作为 Sidecar 与应用容器一起运行在同一个 Pod 中）。



图片 - *Front proxy* 部署结构图（转自 *https://www.envoyproxy.io/docs/envoy/latest/start/sandboxes/front_proxy*）

在此示例中一共有 3 个服务，我们需要为其创建容器编排的 `docker-compose.yml` 文件。

```yaml
version: '2'
services:

  front-envoy:
    build:
      context: .
      dockerfile: Dockerfile-frontenvoy
    volumes:
      - ./front-envoy.yaml:/etc/front-envoy.yaml
    networks:
      - envoymesh
    expose:
      - "80"
      - "8001"
    ports:
      - "8000:80"
      - "8001:8001"

  service1:
    build:
      context: .
      dockerfile: Dockerfile-service
    volumes:
      - ./service-envoy.yaml:/etc/service-envoy.yaml
    networks:
      envoymesh:
        aliases:
          - service1
    environment:
      - SERVICE_NAME=1
    expose:
      - "80"

  service2:
    build:
      context: .
      dockerfile: Dockerfile-service
    volumes:
      - ./service-envoy.yaml:/etc/service-envoy.yaml
    networks:
      envoymesh:
        aliases:
          - service2
    environment:
      - SERVICE_NAME=2
    expose:
      - "80"

networks:
  envoymesh: {}
```

使用 docker-compose 启动可以保证三个服务都在同一个网络内，即 `frontproxy_envoymesh` 网络中。

其中 `front-envoy` 是前端（边缘）Envoy 服务，用来做反向代理，它使用的是 `Dockerfile-frontenvoy` 文件来构建镜像的，我们来看下该 `Dockerfile` 的内容。

```
FROM envoyproxy/envoy:latest

RUN apt-get update && apt-get -q install -y \
    curl
CMD /usr/local/bin/envoy -c /etc/front-envoy.yaml --service-clus
ter front-proxy
```

其中 `/etc/front-envoy.yaml` 是本地的 `front-envoy.yaml` 挂载进去的。我们看下该文件的内容。

```yaml
static_resources:
  listeners:
  - address:
      socket_address:
        address: 0.0.0.0
        port_value: 80
    filter_chains:
    - filters:
      - name: envoy.http_connection_manager
        config:
          codec_type: auto
          stat_prefix: ingress_http
          route_config:
            name: local_route
            virtual_hosts:
            - name: backend
              domains:
              - "*"
              routes:
              - match:
                  prefix: "/service/1"
                route:
                  cluster: service1
              - match:
                  prefix: "/service/2"
                route:
                  cluster: service2
          http_filters:
          - name: envoy.router
            config: {}
  clusters:
```

```yaml
    - name: service1
      connect_timeout: 0.25s
      type: strict_dns
      lb_policy: round_robin
      http2_protocol_options: {}
      hosts:
      - socket_address:
          address: service1
          port_value: 80
    - name: service2
      connect_timeout: 0.25s
      type: strict_dns
      lb_policy: round_robin
      http2_protocol_options: {}
      hosts:
      - socket_address:
          address: service2
          port_value: 80
admin:
  access_log_path: "/dev/null"
  address:
    socket_address:
      address: 0.0.0.0
      port_value: 8001
```

我们看到其中包括了三大配置项：

- **static_resources**：路由配置信息
- **cluster**：envoymesh 的服务注册信息
- **admin**：管理接口，可以通过访问 8001 端口的，访问 `/stats` 获取当前 envoymesh 的一些统计信息，访问 `/server_info` 获取 Envoy 的版本信息

使用 `docker-compose` 启动三个容器。

```
$ pwd
envoy/examples/front-proxy
$ docker-compose up --build -d
$ docker-compose ps
        Name                          Command               State
    Ports
----------------------------------------------------------------
---------------------------------------------
example_service1_1      /bin/sh -c /usr/local/bin/ ... Up
80/tcp
example_service2_1      /bin/sh -c /usr/local/bin/ ... Up
80/tcp
example_front-envoy_1   /bin/sh -c /usr/local/bin/ ... Up
0.0.0.0:8000->80/tcp, 0.0.0.0:8001->8001/tcp
```

我们下面将过一遍 Envoy 作为前端代理的所有功能，这些功能是通用功能。

路由

访问 service1 http://localhost:8000/service/1 将看到如下输出。

```
$ curl -v localhost:8000/service/1
*
Trying ::1...
* TCP_NODELAY set
* Connected to localhost (::1) port 8000 (#0)
> GET /service/1 HTTP/1.1
> Host: localhost:8000
> User-Agent: curl/7.54.0
> Accept: */*
>
< HTTP/1.1 200 OK
< content-type: text/html; charset=utf-8
< content-length: 89
< server: envoy
< date: Fri, 20 Apr 2018 08:26:33 GMT
< x-envoy-upstream-service-time: 14
<
Hello from behind Envoy (service 1)! hostname: a3e4185a9a49 reso
lvedhostname: 172.18.0.4
* Connection #0 to host localhost left intact
```

访问 service2 http://localhost:8000/service/2 将看到如下输出。

```
*   Trying ::1...
* TCP_NODELAY set
* Connected to localhost (::1) port 8000 (#0)
> GET /service/2 HTTP/1.1
> Host: localhost:8000
> User-Agent: curl/7.54.0
> Accept: */*
>
< HTTP/1.1 200 OK
< content-type: text/html; charset=utf-8
< content-length: 89
< server: envoy
< date: Fri, 20 Apr 2018 08:27:27 GMT
< x-envoy-upstream-service-time: 10
<
Hello from behind Envoy (service 2)! hostname: f6650e1911a0 reso
lvedhostname: 172.18.0.3
* Connection #0 to host localhost left intact
```

我们看到访问请求被路由到了正确的服务后端。

负载均衡

增加 service1 的示例数。

```
$ docker-compose scale service1=3
WARNING: The scale command is deprecated. Use the up command wit
h the --scale flag instead.
Starting frontproxy_service1_1 ... done
Creating frontproxy_service1_2 ... done
Creating frontproxy_service1_3 ... done

$ docker-compose ps
          Name                            Command                Stat
e                            Ports
-----------------------------------------------------------------
-----------------------------------------------------------
frontproxy_front-envoy_1   /usr/bin/dumb-init -- /bin ...   Up
    10000/tcp, 0.0.0.0:8000->80/tcp, 0.0.0.0:8001->8001/tcp
frontproxy_service1_1      /bin/sh -c /usr/local/bin/ ...   Up
    10000/tcp, 80/tcp
frontproxy_service1_2      /bin/sh -c /usr/local/bin/ ...   Up
    10000/tcp, 80/tcp
frontproxy_service1_3      /bin/sh -c /usr/local/bin/ ...   Up
    10000/tcp, 80/tcp
frontproxy_service2_1      /bin/sh -c /usr/local/bin/ ...   Up
    10000/tcp, 80/tcp
```

我们看到现在 service1 已经有了 3 个实例，现在再访问 service1 http://localhost:8000/service/1。

```
$ while true;do curl localhost:8000/service/1;sleep 1;done
Hello from behind Envoy (service 1)! hostname: a3e4185a9a49 reso
lvedhostname: 172.18.0.4
Hello from behind Envoy (service 1)! hostname: fe44dba64122 reso
lvedhostname: 172.18.0.5
Hello from behind Envoy (service 1)! hostname: c5b9f1289e0f reso
lvedhostname: 172.18.0.6
Hello from behind Envoy (service 1)! hostname: a3e4185a9a49 reso
lvedhostname: 172.18.0.4
Hello from behind Envoy (service 1)! hostname: fe44dba64122 reso
lvedhostname: 172.18.0.5
Hello from behind Envoy (service 1)! hostname: c5b9f1289e0f reso
lvedhostname: 172.18.0.6
```

我们看到对 service1 的已经有负载均衡了，使用的策略是 `round_robin`，这些都是在 `front-envoy.yaml` 文件中的 `cluster` 项下配置的。

admin 端点

访问 http://localhost:8001 可以看到 Envoy admin 提供以下管理 API 端点。

命令	描述
/	Admin 主页
/certs	打印机器上的 certs
/clusters	upstream cluster 状态
/config_dump	输出当前的 Envoy 配置
/cpuprofiler	开启/关闭 CPU profiler
/healthcheck/fail	导致服务失败健康检查
/healthcheck/ok	导致服务通过健康检查
/help	打印管理命令的帮助信息
/hot_restart_version	打印热重启兼容版本
/listeners	打印 listener 地址
/logging	查询/更改日志级别
/quitquitquit	退出服务
/reset_counters	将计数器重置为 1
/runtime	打印运行时值
/runtime_modify	修改运行时值
/server_info	打印服务器版本/状态信息
/stats	打印服务器状态统计信息
/stats/prometheus	打印 prometheus 格式的服务器状态统计信息

Envoy 提供了 API 管理端点，可以对 Envoy 进行动态配置，参考 v2 API reference。

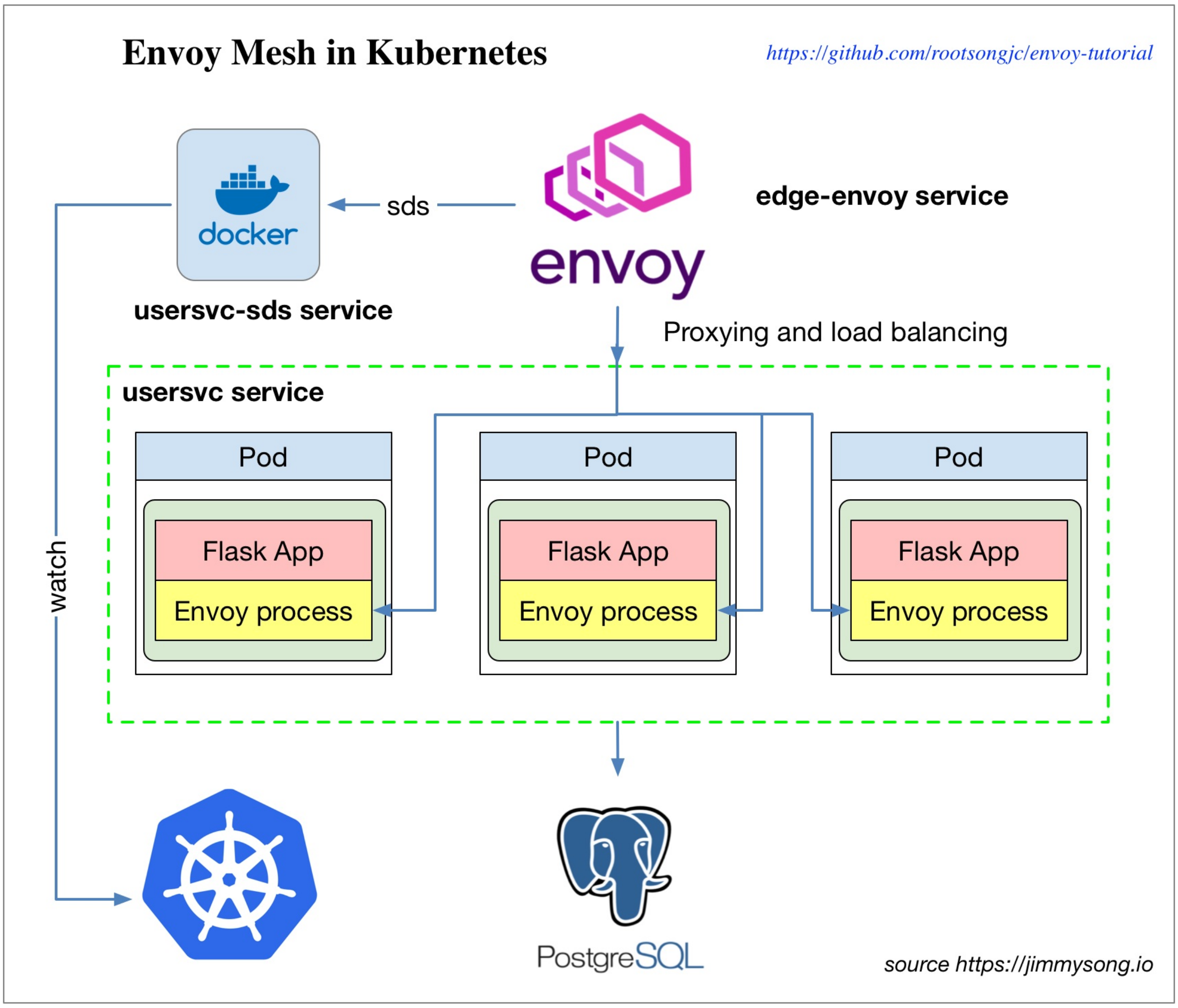
参考

- Front proxy

Envoy mesh 教程

本文是在 Kubernetes 集群中，使用 Envoy 来做 mesh，来为一个简单的使用 Python 编写的 Flask 应用程序做反向代理和负载均衡。

注：本教程中的示例来自 envoy-steps，本文中使用的所有的代码和 YAML 配置见 envoy-tutorial。



图片 - *Envoy Mesh*架构图

前提条件

使用 kubernetes-vagrant-centos-cluster 部署 kubernetes 集群，只要启动集群并安装了 CoreDNS 即可，无须安装其他插件。

部署应用

我们首先将应用部署到 Kubernetes 中。

部署 postgres 数据库。

```
kubectl apply -f postgres
```

创建 usersvc 镜像。

```
docker build -t jimmysong/usersvc:step1 .
```

部署 usersvc。

```
kubectl apply -f usersvc
```

查看 uservc 的 ClusterIP 地址。

```
$ kubectl get svc usersvc
kubectl get svc usersvc
NAME       TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)
AGE
usersvc    ClusterIP   10.254.176.248   <none>        5000/TCP
11m
```

进到 node1 中访问该服务，因为我们要访问的是 ClusterIP，在我们自己的电脑上是无法直接访问的，所以进到虚拟机中操作。

```
$ vagrant ssh node1
$ curl 10.254.176.248:5000
{
  "hostname": "usersvc-7cf5bb9d85-9gx7w",
  "msg": "user health check OK",
  "ok": true,
  "resolvedname": "172.33.10.7"
}
```

尝试添加一个名为 `Alice` 的用户。

```
$ curl -X PUT -H "Content-Type: application/json" \
    -d '{ "fullname": "Alice", "password": "alicerules" }' \
    10.254.176.248/user/alice
```

将会看到类似如下的输出。

```
{
  "fullname": "Alice",
  "hostname": "usersvc-7cf5bb9d85-9gx7w",
  "ok": true,
  "resolvedname": "172.33.10.7",
  "uuid": "EF43B475F65848C6BE708F436305864B"
}
```

尝试再添加一个名为 `Bob` 的用户。

```
$ curl -X PUT -H "Content-Type: application/json" \
    -d '{ "fullname": "Bob", "password": "bobrules" }' \
    10.254.176.248/user/bob
```

将会看到类似如下的输出。

```
{
  "fullname": "Bob",
  "hostname": "usersvc-7cf5bb9d85-9gx7w",
  "ok": true,
  "resolvedname": "172.33.10.7",
  "uuid": "6AC944E7D4254D9A811A82C0FDAC3046"
}
```

当应用部署完毕后，我们该部署 edge envoy 了。

部署 edge envoy

部署 edge envoy 的方式很简单，执行下面的命令。

```
kubectl apply -f edge-envoy
```

现在访问 edge envoy 是就可以路由到 `usersvc` 上的，当然直接访问 `usersvc` 也是可以的。

我们看下 edge-envoy 的 envoy 配置文件定义。

```
{
  "listeners": [
    {
      "address": "tcp://0.0.0.0:80",
      "filters": [
        {
          "type": "read",
          "name": "http_connection_manager",
          "config": {
            "codec_type": "auto",
            "stat_prefix": "ingress_http",
            "route_config": {
              "virtual_hosts": [
                {
                  "name": "backend",
                  "domains": ["*"],
                  "routes": [
                    {
                      "timeout_ms": 0,
                      "prefix": "/user",
                      "cluster": "usersvc"
                    }
                  ]
                }
              ]
            },
            "filters": [
              {
                "type": "decoder",
                "name": "router",
                "config": {}
              }
            ]
          }
        }
      ]
    }
  ],
  "admin": {
    "access_log_path": "/dev/null",
    "address": "tcp://127.0.0.1:8001"
  },
  "cluster_manager": {
    "clusters": [
      {
        "name": "usersvc",
```

```
            "connect_timeout_ms": 250,
            "type": "strict_dns",
            "service_name": "usersvc",
            "lb_type": "round_robin",
            "features": "http2",
            "hosts": [
              {
                "url": "tcp://usersvc:80"
              }
            ]
          }
        ]
      }
    }
```

客户端访问 `edge-envoy` 的 `ClusterIP:8000/user/health` 就可以检查节点的健康状况。

部署 usersvc2

删除原来的 `usersvc`，部署第二版 `usersvc2`，它与原来的 `usersvc` 唯一不同的地方是在 `entrypoint` 中集成了 envoy，查看 `Dockerfile` 中指定的 `entrypoint.sh` 的内容便可知。

```
#!/bin/sh

python /application/service.py &
/usr/local/bin/envoy -c /application/envoy.json
```

首先删除老的 `usersvc`。

```
kubectl delete -f usersvc
```

使用下面的命令部署 `usersvc2`，它仍然使用 `usersvc` 这个 service 名称。

```
kubectl apply -f usersvc2
```

Envoy 以 out-of-process 的方式运行，对应用进程没有侵入性，也可以使用 sidecar 的方式运行，让 envoy 与 应用容器运行在同一个 pod 中。

增加 `usersvc2` 的实例个数。

```
kubectl scale --replicas=3 deployment/usersvc
```

此时我们有 3 个 usersvc 实例，现在通过 `edge-envoy` 的 `ClusterIP:8000/user/health` 检查节点的健康状况时，是不是会轮询的访问到后端的的 `usersvc2` 的实例呢？

我们当初在 `edge-node` 的 `envoy.json` 中配置过 cluster 的，其中指定了 `lb_type` 为 `round_robin` 。

```
"cluster_manager": {
  "clusters": [
    {
      "name": "usersvc",
      "connect_timeout_ms": 250,
      "type": "strict_dns",
      "service_name": "usersvc",
      "lb_type": "round_robin",
      "features": "http2",
      "hosts": [
        {
          "url": "tcp://usersvc:80"
        }
      ]
    }
  ]
}
```

而且该 `serivce_name` 也可以被 DNS 正确解析。

```
root@usersvc-55b6857d44-gcg5c:/application# nslookup usersvc
Server:         10.254.0.2
Address:        10.254.0.2#53

Name:    usersvc.envoy-tutorial.svc.cluster.local
Address: 10.254.123.166
```

答案是否定的。

虽然通过 DNS 可以正确的解析出 serivce 的 ClusterIP，但是负载均衡不再通过 kube-proxy 实现，所以不论我们访问多少次 `edge-envoy` 永远只能访问到一个固定的后端 `usersvc` 。

服务发现服务 - SDS

Kubernetes 中的 DNS 可以发现所有 serivce 的 ClusterIP，但是 DNS 中不包括所有 endpoint 地址，我们需要一个 SDS（服务发现服务）来发现服务的所有的 endpoint，我们将修改 `lb_type` ，使用 `sds` 替代 `strict_dns` 。

执行下面的命令部署 SDS。

```
kubectl apply -f usersvc-sds
```

因为在添加了 SDS 之后需要修改 `edge-envoy` 中的 `envoy.josn` 配置，在 `clusters` 字段中增加 `sds` 信息，我们将所有的配置都写好了，重新打包成了镜像，我们需要先删除之前部署的 `edge-envoy` 。

```
kubectl delete -f edge-envoy
```

部署新的 `edge-envoy2` 。

```
kubectl apply -f edge-envoy2
```

连续访问 `usersvc` 12 次看看输出结果如何。

```
URL=http://172.17.8.101:30800/user/alice
for i in `seq 1 12`;do curl -s $URL|grep "resolvedname"|tr -d " "
|tr -d ","|tr -d '"';done
```

我们可以看到类似如下的输出：

```
resolvedname:172.33.71.2
resolvedname:172.33.88.2
resolvedname:172.33.10.2
resolvedname:172.33.71.2
resolvedname:172.33.88.2
resolvedname:172.33.10.2
resolvedname:172.33.71.2
resolvedname:172.33.88.2
resolvedname:172.33.10.2
resolvedname:172.33.71.2
resolvedname:172.33.88.2
resolvedname:172.33.10.2
```

再查看下 `usersvc` 服务的所有 pod 的 IP 地址。

```
$ kubectl get pod -l service=usersvc -o wide
NAME                          READY     STATUS    RESTARTS   AGE
    IP              NODE
usersvc-55b6857d44-mkfpv   1/1       Running   0          9m
    172.33.88.2    node1
usersvc-55b6857d44-q98jg   1/1       Running   0          9m
    172.33.71.2    node2
usersvc-55b6857d44-s2znk   1/1       Running   0          9m
    172.33.10.2    node3
```

我们看到 round-robin 负载均衡生效了。

参考

- Part 2: Deploying Envoy with a Python Flask webapp and Kubernetes
- envoy-steps
- kubernetes-vagrant-centos-cluster
- envoy-tutorial

大数据

Kubernetes community中已经有了一个Big data SIG，大家可以通过这个SIG了解kubernetes结合大数据的应用。

在Swarm、Mesos、kubernetes这三种流行的容器编排调度架构中，Mesos对于大数据应用支持是最好的，spark原生就是运行在mesos上的，当然也可以容器化运行在kubernetes上。当前在kubernetes上运行大数据应用主要是spark应用。

Spark on Kubernetes

Spark原生支持standalone、mesos和YARN的调度方式，当前kubernetes社区正在支持kubernetes的原生调度来运行spark - 。

当然您也可以在kubernetes直接部署spark on yarn或者spark standalone模式，仍然沿用已有的

Spark Standalone

使用spark standalone模式在kubernetes上运行，kubernetes不负责spark任务的调度。参考：Spark standalone on Kubernetes

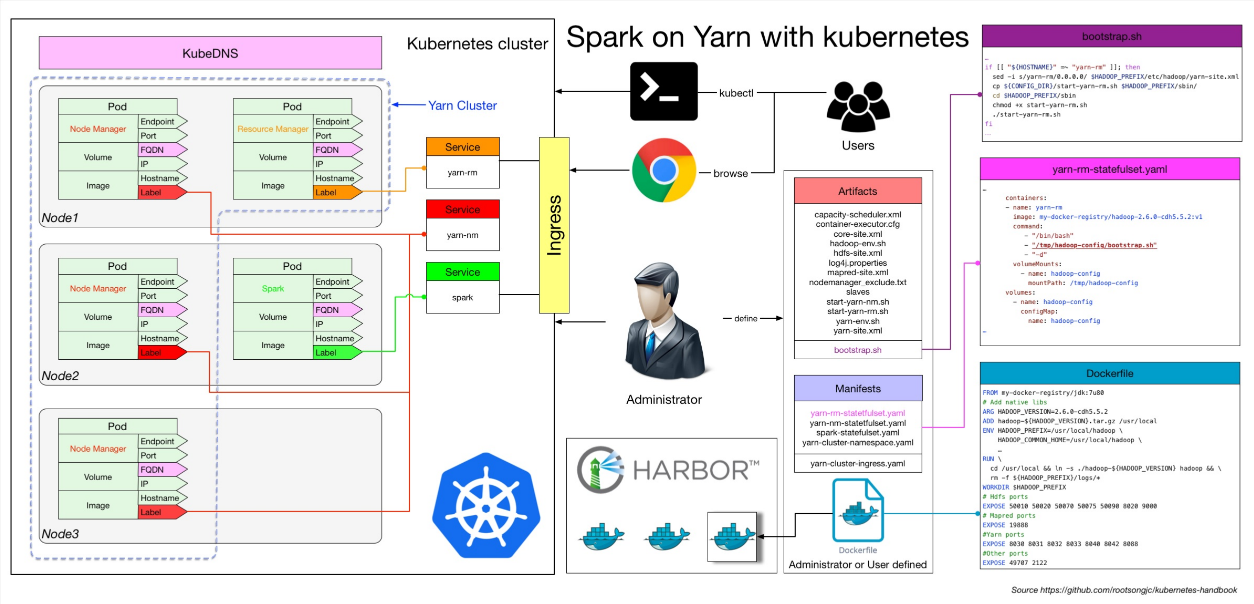
这种模式中使用的spark本身负责任务调度，kubernetes只是作为一个spark的部署平台。

Spark on Yarn

使用StatefulSet和Headless serverless来实现，请参考 Spark on Yarn

这种模式中kubernetes依然不负责spark应用的调度，而只是将Yarn换了一个部署环境而已。

下面是架构图：



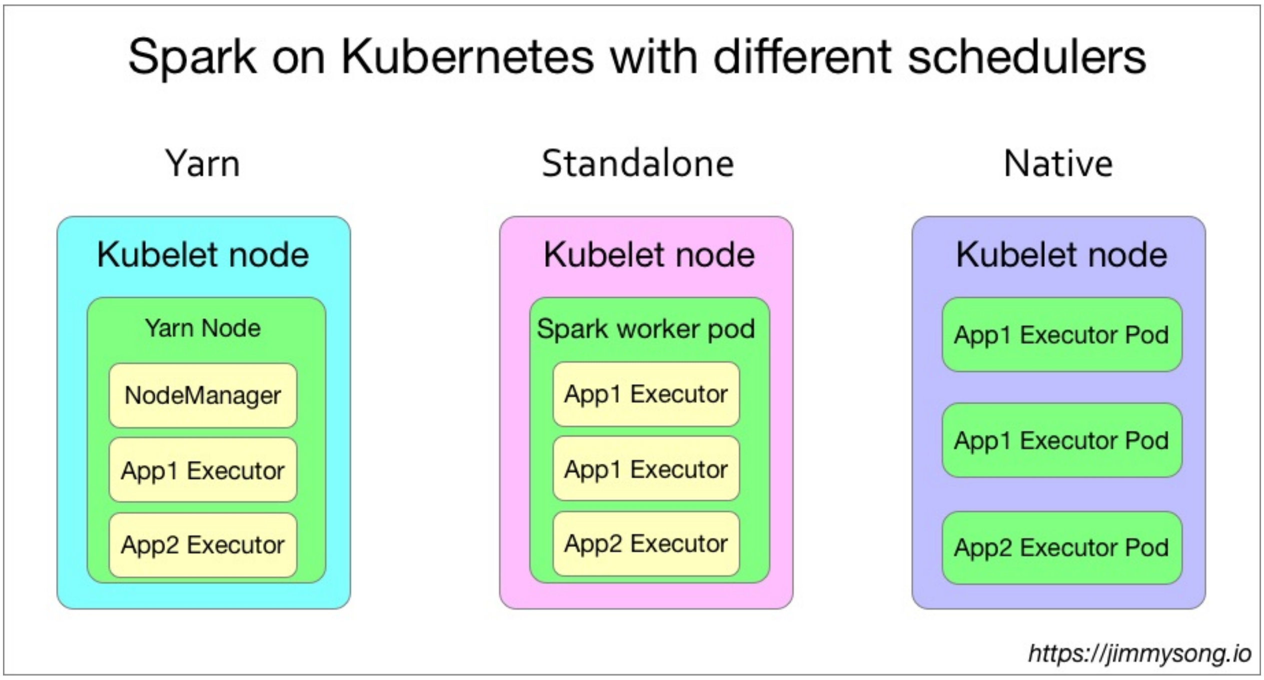
图片 - *Spark on yarn with kubernetes*

Spark on Kubernetes

Spark on kubernetes，使用kubernetes作为调度引擎，spark的任务直接调度到node节点上。参考：[运行支持kubernetes原生调度的Spark程序]

调度方式总结

下图显示的是三种调度方式中单个kubernetes node节点上运行的spark相关容器的调度情况。



图片 - 在*kubernetes*上使用多种调度方式

毫无疑问，使用kubernetes原生调度的spark任务才是最节省资源的。

Spark standalone on Kubernetes

该项目是基于 Spark standalone 模式，对资源的分配调度还有作业状态查询的功能实在有限，对于让 spark 使用真正原生的 kubernetes 资源调度推荐大家尝试 https://github.com/apache-spark-on-k8s/

本文中使用的镜像我已编译好上传到了时速云上，大家可以直接下载。

```
index.tenxcloud.com/jimmy/spark:1.5.2_v1
index.tenxcloud.com/jimmy/zeppelin:0.7.1
```

代码和使用文档见Github地址：https://github.com/rootsongjc/spark-on-kubernetes

本文中用到的 yaml 文件可以在 ../manifests/spark-standalone 目录下找到，也可以在上面的 https://github.com/rootsongjc/spark-on-kubernetes/ 项目的 manifests 目录下找到。

注意：时速云上本来已经提供的镜像 `index.tenxcloud.com/google_containers/spark:1.5.2_v1` ，但是该镜像似乎有问题，下载总是失败。

在Kubernetes上启动spark

创建名为spark-cluster的namespace，所有操作都在该namespace中进行。

所有yaml文件都在 `manifests` 目录下。

```
$ kubectl create -f manifests/
```

将会启动一个拥有三个worker的spark集群和zeppelin。

同时在该namespace中增加ingress配置，将spark的UI和zeppelin页面都暴露出来，可以在集群外部访问。

该ingress后端使用traefik。

访问spark

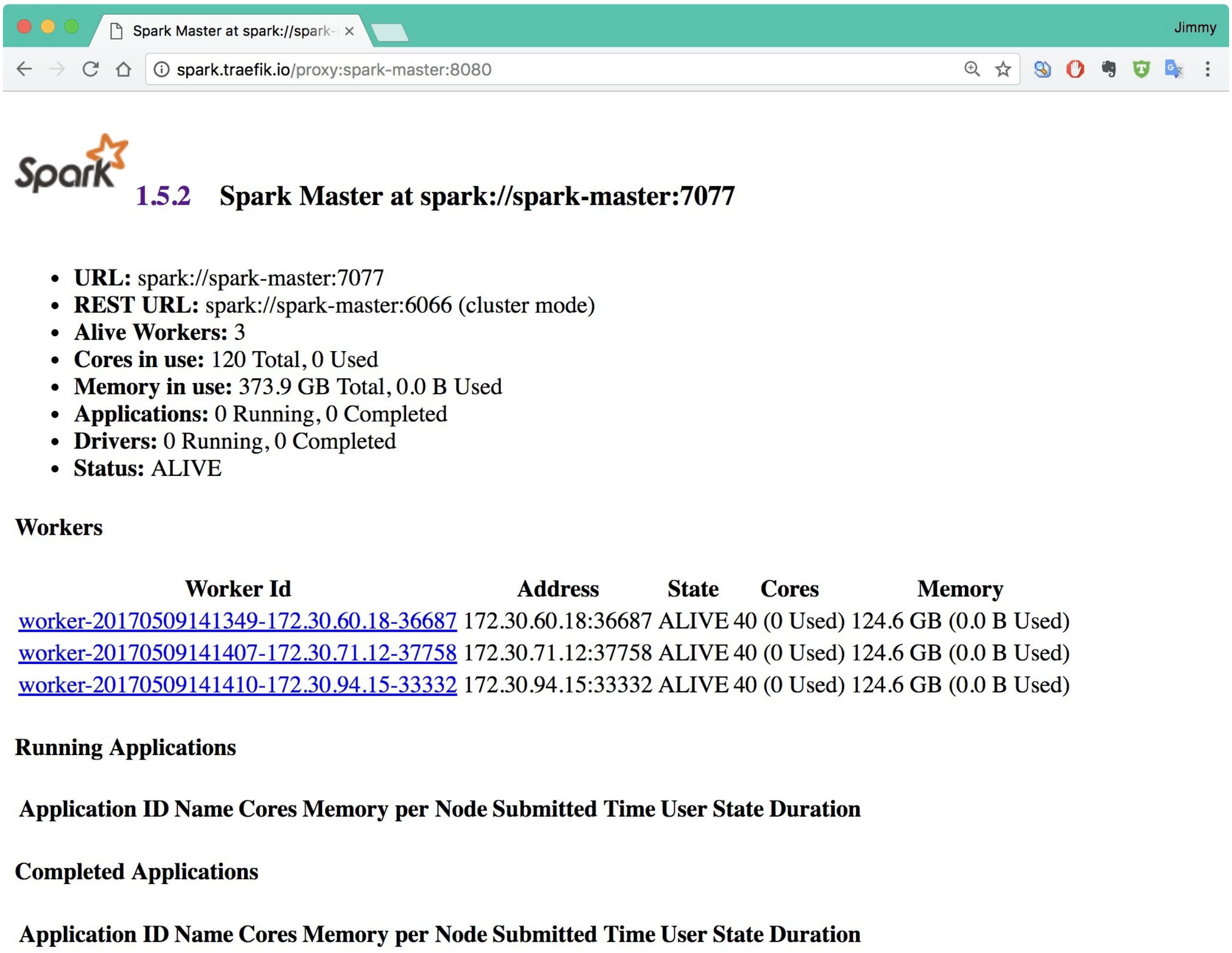
通过上面对ingress的配置暴露服务，需要修改本机的/etc/hosts文件，增加以下配置，使其能够解析到上述service。

```
172.20.0.119 zeppelin.traefik.io
172.20.0.119 spark.traefik.io
```

172.20.0.119是我设置的VIP地址，VIP的设置和traefik的配置请查看kubernetes-handbook。

spark ui

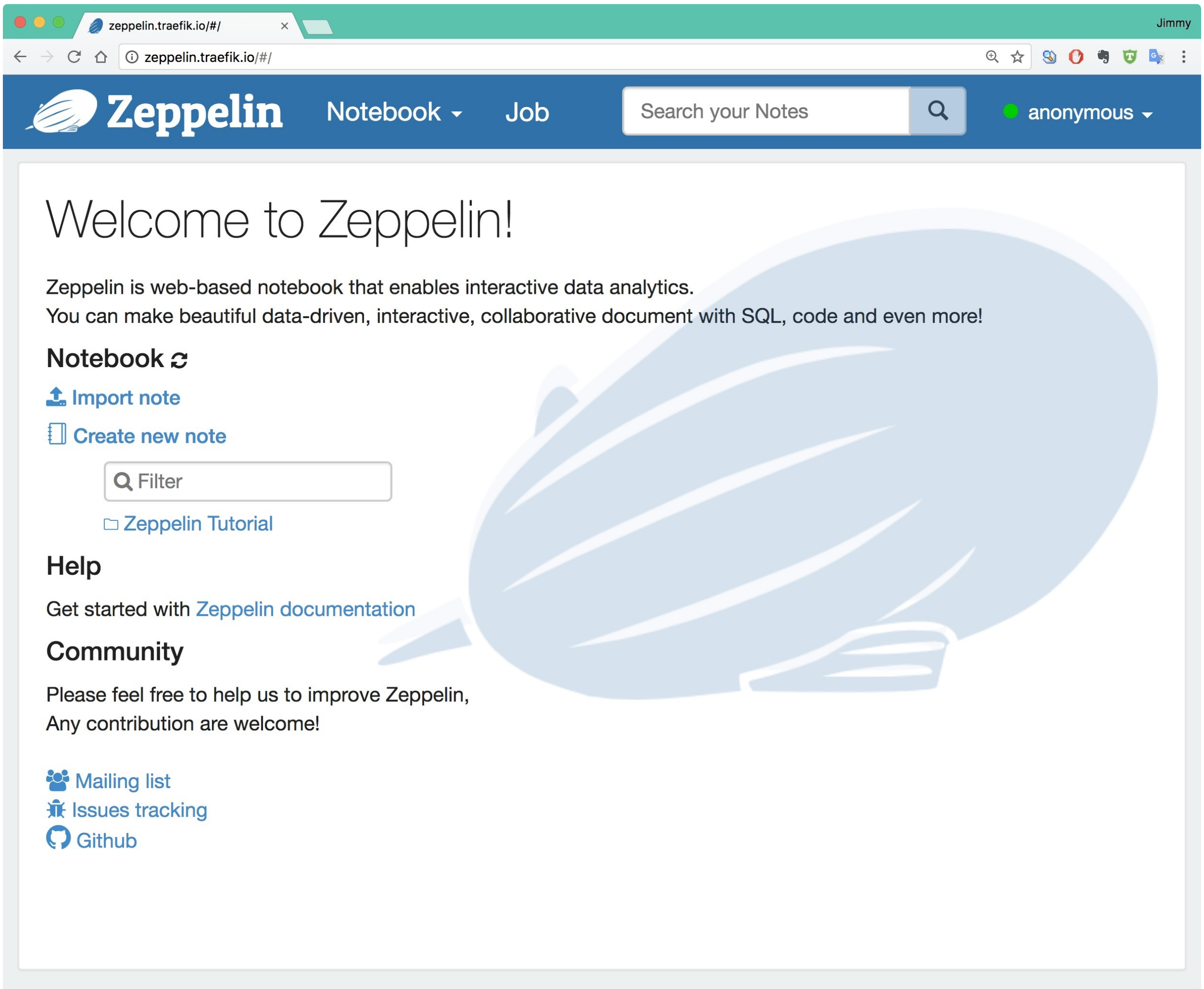
访问http://spark.traefik.io



图片 - *spark master ui*

zeppelin ui

访问http://zepellin.treafik.io



运行支持kubernetes原生调度的Spark程序

TL;DR 这个主题比较大，该开源项目也还在不断进行中，我单独做了一个 web 用来记录 spark on kubernetes 的研究和最新进展见: https://jimmysong.io/spark-on-k8s

我们之前就在 kubernetes 中运行过 standalone 方式的 spark 集群，见 Spark standalone on kubernetes。

目前运行支持 kubernetes 原生调度的 spark 程序由 Google 主导，目前运行支持 kubernetes 原生调度的 spark 程序由 Google 主导，fork 自 spark 的官方代码库，见https://github.com/apache-spark-on-k8s/spark/ ，属于Big Data SIG。

参与到该项目的公司有：

- Bloomberg
- Google
- Haiwen
- Hyperpilot
- Intel
- Palantir
- Pepperdata
- Red Hat

为何使用 spark on kubernetes

使用kubernetes原生调度的spark on kubernetes是对现有的spark on yarn/mesos的资源使用方式的革命性的改进，主要表现在以下几点：

1. Kubernetes原生调度：不再需要二层调度，直接使用kubernetes的资源调度功能，跟其他应用共用整个kubernetes管理的资源池；
2. 资源隔离，粒度更细：原先yarn中的queue在spark on kubernetes中已不存在，取而代之的是kubernetes中原生的namespace，可以为每个用户分别指定一个namespace，限制用户的资源quota；
3. 细粒度的资源分配：可以给每个spark任务指定资源限制，实际指定多少资源就使用多少资源，因为没有了像yarn那样的二层调度（圈地式的），所以可以

更高效和细粒度的使用资源；
4. 监控的变革：因为做到了细粒度的资源分配，所以可以对用户提交的每一个任务做到资源使用的监控，从而判断用户的资源使用情况，所有的metric都记录在数据库中，甚至可以为每个用户的每次任务提交计量；
5. 日志的变革：用户不再通过yarn的web页面来查看任务状态，而是通过pod的log来查看，可将所有的kuberentes中的应用的日志等同看待收集起来，然后可以根据标签查看对应应用的日志；

所有这些变革都可以让我们更高效的获取资源、更有效率的获取资源！

Spark 概念说明

Apache Spark 是一个围绕速度、易用性和复杂分析构建的大数据处理框架。最初在2009年由加州大学伯克利分校的AMPLab开发，并于2010年成为Apache的开源项目之一。

在 Spark 中包括如下组件或概念：

- **Application**：Spark Application 的概念和 Hadoop 中的 MapReduce 类似，指的是用户编写的 Spark 应用程序，包含了一个 Driver 功能的代码和分布在集群中多个节点上运行的 Executor 代码；
- **Driver**：Spark 中的 Driver 即运行上述 Application 的 main() 函数并且创建 SparkContext，其中创建 SparkContext 的目的是为了准备Spark应用程序的运行环境。在 Spark 中由 SparkContext 负责和 ClusterManager 通信，进行资源的申请、任务的分配和监控等；当 Executor 部分运行完毕后，Driver负责将 SparkContext 关闭。通常用 SparkContext 代表 Driver；
- **Executor**：Application运行在Worker 节点上的一个进程，该进程负责运行Task，并且负责将数据存在内存或者磁盘上，每个Application都有各自独立的一批Executor。在Spark on Yarn模式下，其进程名称为 `CoarseGrainedExecutorBackend` ，类似于 Hadoop MapReduce 中的 YarnChild。一个 `CoarseGrainedExecutorBackend` 进程有且仅有一个 executor 对象，它负责将 Task 包装成 taskRunner，并从线程池中抽取出一个空闲线程运行 Task。每个 `CoarseGrainedExecutorBackend` 能并行运行 Task 的数量就取决于分配给它的 CPU 的个数了；
- **Cluster Manager**：指的是在集群上获取资源的外部服务，目前有：
  - Standalone：Spark原生的资源管理，由Master负责资源的分配；
  - Hadoop Yarn：由YARN中的ResourceManager负责资源的分配；

- **Worker**：集群中任何可以运行Application代码的节点，类似于YARN中的NodeManager节点。在Standalone模式中指的就是通过Slave文件配置的Worker节点，在Spark on Yarn模式中指的就是NodeManager节点；
- 作业（**Job**）：包含多个Task组成的并行计算，往往由Spark Action催生，一个JOB包含多个RDD及作用于相应RDD上的各种Operation；
- 阶段（**Stage**）：每个Job会被拆分很多组 Task，每组任务被称为Stage，也可称TaskSet，一个作业分为多个阶段，每一个stage的分割点是action。比如一个job是：（transformation1 -> transformation1 -> action1 -> transformation3 -> action2），这个job就会被分为两个stage，分割点是action1和action2。
- 任务（**Task**）： 被送到某个Executor上的工作任务；
- **Context**：启动spark application的时候创建，作为Spark 运行时环境。
- **Dynamic Allocation**（动态资源分配）：一个配置选项，可以将其打开。从Spark1.2之后，对于On Yarn模式，已经支持动态资源分配（Dynamic Resource Allocation），这样，就可以根据Application的负载（Task情况），动态的增加和减少executors，这种策略非常适合在YARN上使用spark-sql做数据开发和分析，以及将spark-sql作为长服务来使用的场景。Executor 的动态分配需要在 cluster mode 下启用 "external shuffle service"。
- 动态资源分配策略：开启动态分配策略后，application会在task因没有足够资源被挂起的时候去动态申请资源，这意味着该application现有的executor无法满足所有task并行运行。spark一轮一轮的申请资源，当有task挂起或等待 `spark.dynamicAllocation.schedulerBacklogTimeout` (默认1s)时间的时候，会开始动态资源分配；之后会每隔 `spark.dynamicAllocation.sustainedSchedulerBacklogTimeout` (默认1s)时间申请一次，直到申请到足够的资源。每次申请的资源量是指数增长的，即1,2,4,8等。之所以采用指数增长，出于两方面考虑：其一，开始申请的少是考虑到可能application会马上得到满足；其次要成倍增加，是为了防止application需要很多资源，而该方式可以在很少次数的申请之后得到满足。

架构设计

关于 spark standalone 的局限性与 kubernetes native spark 架构之间的区别请参考 Anirudh Ramanathan 在 2016年10月8日提交的 issue Support Spark natively in Kubernetes #34377。

简而言之，spark standalone on kubernetes 有如下几个缺点：

- 无法对于多租户做隔离，每个用户都想给 pod 申请 node 节点可用的最大的资源。
- Spark 的 master／worker 本来不是设计成使用 kubernetes 的资源调度，这样会存在两层的资源调度问题，不利于与 kuberentes 集成。

而 kubernetes native spark 集群中，spark 可以调用 kubernetes API 获取集群资源和调度。要实现 kubernetes native spark 需要为 spark 提供一个集群外部的 manager 可以用来跟 kubernetes API 交互。

调度器后台

使用 kubernetes 原生调度的 spark 的基本设计思路是将 spark 的 driver 和 executor 都放在 kubernetes 的 pod 中运行，另外还有两个附加的组件： `ResourceStagingServer` 和 `KubernetesExternalShuffleService` 。

Spark driver 其实可以运行在 kubernetes 集群内部（cluster mode）可以运行在外部（client mode），executor 只能运行在集群内部，当有 spark 作业提交到 kubernetes 集群上时，调度器后台将会为 executor pod 设置如下属性：

- 使用我们预先编译好的包含 kubernetes 支持的 spark 镜像，然后调用 `CoarseGrainedExecutorBackend` main class 启动 JVM。
- 调度器后台为 executor pod 的运行时注入环境变量，例如各种 JVM 参数，包括用户在 `spark-submit` 时指定的那些参数。
- Executor 的 CPU、内存限制根据这些注入的环境变量保存到应用程序的 `SparkConf` 中。
- 可以在配置中指定 spark 运行在指定的 namespace 中。

参考：Scheduler backend 文档

安装指南

我们可以直接使用官方已编译好的 docker 镜像来部署，下面是官方发布的镜像：

组件	镜像
Spark Driver Image	`kubespark/spark-driver:v2.1.0-kubernetes-0.3.1`
Spark Executor Image	`kubespark/spark-executor:v2.1.0-kubernetes-0.3.1`
Spark Initialization Image	`kubespark/spark-init:v2.1.0-kubernetes-0.3.1`
Spark Staging Server Image	`kubespark/spark-resource-staging-server:v2.1.0-kubernetes-0.3.1`
PySpark Driver Image	`kubespark/driver-py:v2.1.0-kubernetes-0.3.1`
PySpark Executor Image	`kubespark/executor-py:v2.1.0-kubernetes-0.3.1`

我将这些镜像放到了我的私有镜像仓库中了。

还需要安装支持 kubernetes 的 spark 客户端，在这里下载：https://github.com/apache-spark-on-k8s/spark/releases

根据使用的镜像版本，我下载的是 v2.1.0-kubernetes-0.3.1

运行 **SparkPi** 测试

我们将任务运行在 `spark-cluster` 的 namespace 中，启动 5 个 executor 实例。

```
./bin/spark-submit \
  --deploy-mode cluster \
  --class org.apache.spark.examples.SparkPi \
  --master k8s://https://172.20.0.113:6443 \
  --kubernetes-namespace spark-cluster \
  --conf spark.executor.instances=5 \
  --conf spark.app.name=spark-pi \
  --conf spark.kubernetes.driver.docker.image=harbor-001.jimmysong.io/library/kubespark-spark-driver:v2.1.0-kubernetes-0.3.1 \
  --conf spark.kubernetes.executor.docker.image=harbor-001.jimmysong.io/library/kubespark-spark-executor:v2.1.0-kubernetes-0.3.1 \
  --conf spark.kubernetes.initcontainer.docker.image=harbor-001.jimmysong.io/library/kubespark-spark-init:v2.1.0-kubernetes-0.3.1 \
local:///opt/spark/examples/jars/spark-examples_2.11-2.1.0-k8s-0.3.1-SNAPSHOT.jar
```

关于该命令参数的介绍请参考：https://apache-spark-on-k8s.github.io/userdocs/running-on-kubernetes.html

注意：该 jar 包实际上是 `spark.kubernetes.executor.docker.image` 镜像中的。

这时候提交任务运行还是失败，报错信息中可以看到两个问题：

- Executor 无法找到 driver pod
- 用户 `system:serviceaccount:spark-cluster:defaul` 没有权限获取 `spark-cluster` 中的 pod 信息。

提了个 issue Failed to run the sample spark-pi test using spark-submit on the doc #478

需要为 spark 集群创建一个 `serviceaccount` 和 `clusterrolebinding`：

```
kubectl create serviceaccount spark --namespace spark-cluster
kubectl create rolebinding spark-edit --clusterrole=edit --servi
ceaccount=spark-cluster:spark --namespace=spark-cluster
```

该 Bug 将在新版本中修复。

用户指南

编译

Fork 并克隆项目到本地：

```
git clone https://github.com/rootsongjc/spark.git
```

编译前请确保你的环境中已经安装 Java8 和 Maven3。

```
## 第一次编译前需要安装依赖
build/mvn install -Pkubernetes -pl resource-managers/kubernetes/core -am -DskipTests

## 编译 spark on kubernetes
build/mvn compile -Pkubernetes -pl resource-managers/kubernetes/core -am -DskipTests

## 发布
dev/make-distribution.sh --tgz -Phadoop-2.7 -Pkubernetes
```

第一次编译和发布的过程耗时可能会比较长，请耐心等待，如果有依赖下载不下来，请自备梯子。

详细的开发指南请见：https://github.com/apache-spark-on-k8s/spark/blob/branch-2.2-kubernetes/resource-managers/kubernetes/README.md

构建镜像

使用该脚本来自动构建容器镜像：https://github.com/apache-spark-on-k8s/spark/pull/488

将该脚本放在 `dist` 目录下，执行：

```
./build-push-docker-images.sh -r harbor-001.jimmysong.io/library -t v2.1.0-kubernetes-0.3.1-1 build
./build-push-docker-images.sh -r harbor-001.jimmysong.io/library -t v2.1.0-kubernetes-0.3.1-1 push
```

注意：如果你使用的 MacOS，bash 的版本可能太低，执行改脚本将出错，请检查你的 bash 版本：

```
bash --version
GNU bash, version 3.2.57(1)-release (x86_64-apple-darwin16)
Copyright (C) 2007 Free Software Foundation, Inc.
```

上面我在升级 bash 之前获取的版本信息，使用下面的命令升级 bash：

```
brew install bash
```

升级后的 bash 版本为 `4.4.12(1)-release (x86_64-apple-darwin16.3.0)`。

编译并上传镜像到我的私有镜像仓库，将会构建出如下几个镜像：

```
harbor-001.jimmysong.io/library/spark-driver:v2.1.0-kubernetes-0
.3.1-1
harbor-001.jimmysong.io/library/spark-resource-staging-server:v2
.1.0-kubernetes-0.3.1-1
harbor-001.jimmysong.io/library/spark-init:v2.1.0-kubernetes-0.3
.1-1
harbor-001.jimmysong.io/library/spark-shuffle:v2.1.0-kubernetes-
0.3.1-1
harbor-001.jimmysong.io/library/spark-executor:v2.1.0-kubernetes
-0.3.1-1
harbor-001.jimmysong.io/library/spark-executor-py:v2.1.0-kuberne
tes-0.3.1-1
harbor-001.jimmysong.io/library/spark-driver-py:v2.1.0-kubernete
s-0.3.1-1
```

运行测试

在 `dist/bin` 目录下执行 spark-pi 测试：

```
./spark-submit \
  --deploy-mode cluster \
  --class org.apache.spark.examples.SparkPi \
  --master k8s://https://172.20.0.113:6443 \
  --kubernetes-namespace spark-cluster \
  --conf spark.kubernetes.authenticate.driver.serviceAccountName=
spark \
  --conf spark.executor.instances=5 \
  --conf spark.app.name=spark-pi \
  --conf spark.kubernetes.driver.docker.image=harbor-001.jimmyso
ng.io/library/spark-driver:v2.1.0-kubernetes-0.3.1-1 \
  --conf spark.kubernetes.executor.docker.image=harbor-001.jimmy
song.io/library/spark-executor:v2.1.0-kubernetes-0.3.1-1 \
  --conf spark.kubernetes.initcontainer.docker.image=harbor-001.
jimmysong.io/library/spark-init:v2.1.0-kubernetes-0.3.1-1 \
local:///opt/spark/examples/jars/spark-examples_2.11-2.2.0-k8s-0
.4.0-SNAPSHOT.jar
```

详细的参数说明见：https://apache-spark-on-k8s.github.io/userdocs/running-on-kubernetes.html

注意：`local:///opt/spark/examples/jars/spark-examples_2.11-2.2.0-k8s-0.4.0-SNAPSHOT.jar` 文件是在 `spark-driver` 和 `spark-executor` 镜像里的，在上一步构建镜像时已经构建并上传到了镜像仓库中。

执行日志显示：

```
2017-09-14 14:59:01 INFO  Client:54 - Waiting for application sp
ark-pi to finish...
2017-09-14 14:59:01 INFO  LoggingPodStatusWatcherImpl:54 - State
 changed, new state:
     pod name: spark-pi-1505372339796-driver
     namespace: spark-cluster
     labels: spark-app-selector -> spark-f4d3a5d3ad964a05a51feb6
191d50357, spark-role -> driver
     pod uid: 304cf440-991a-11e7-970c-f4e9d49f8ed0
     creation time: 2017-09-14T06:59:01Z
     service account name: spark
     volumes: spark-token-zr8wv
     node name: N/A
     start time: N/A
     container images: N/A
     phase: Pending
     status: []
2017-09-14 14:59:01 INFO  LoggingPodStatusWatcherImpl:54 - State
 changed, new state:
     pod name: spark-pi-1505372339796-driver
     namespace: spark-cluster
     labels: spark-app-selector -> spark-f4d3a5d3ad964a05a51feb6
191d50357, spark-role -> driver
     pod uid: 304cf440-991a-11e7-970c-f4e9d49f8ed0
     creation time: 2017-09-14T06:59:01Z
     service account name: spark
     volumes: spark-token-zr8wv
     node name: 172.20.0.114
     start time: N/A
     container images: N/A
     phase: Pending
     status: []
2017-09-14 14:59:01 INFO  LoggingPodStatusWatcherImpl:54 - State
 changed, new state:
     pod name: spark-pi-1505372339796-driver
     namespace: spark-cluster
     labels: spark-app-selector -> spark-f4d3a5d3ad964a05a51feb6
191d50357, spark-role -> driver
     pod uid: 304cf440-991a-11e7-970c-f4e9d49f8ed0
     creation time: 2017-09-14T06:59:01Z
     service account name: spark
     volumes: spark-token-zr8wv
     node name: 172.20.0.114
     start time: 2017-09-14T06:59:01Z
     container images: harbor-001.jimmysong.io/library/spark-dri
```

```
ver:v2.1.0-kubernetes-0.3.1-1
     phase: Pending
     status: [ContainerStatus(containerID=null, image=harbor-001
.jimmysong.io/library/spark-driver:v2.1.0-kubernetes-0.3.1-1, im
ageID=, lastState=ContainerState(running=null, terminated=null,
waiting=null, additionalProperties={}), name=spark-kubernetes-dr
iver, ready=false, restartCount=0, state=ContainerState(running=
null, terminated=null, waiting=ContainerStateWaiting(message=nul
l, reason=ContainerCreating, additionalProperties={}), additiona
lProperties={}), additionalProperties={})]
2017-09-14 14:59:03 INFO  LoggingPodStatusWatcherImpl:54 - State
 changed, new state:
     pod name: spark-pi-1505372339796-driver
     namespace: spark-cluster
     labels: spark-app-selector -> spark-f4d3a5d3ad964a05a51feb6
191d50357, spark-role -> driver
     pod uid: 304cf440-991a-11e7-970c-f4e9d49f8ed0
     creation time: 2017-09-14T06:59:01Z
     service account name: spark
     volumes: spark-token-zr8wv
     node name: 172.20.0.114
     start time: 2017-09-14T06:59:01Z
     container images: harbor-001.jimmysong.io/library/spark-dri
ver:v2.1.0-kubernetes-0.3.1-1
     phase: Running
     status: [ContainerStatus(containerID=docker://5c5c821c482a1
e35552adccb567020532b79244392374f25754f0050e6cd4c62, image=harbo
r-001.jimmysong.io/library/spark-driver:v2.1.0-kubernetes-0.3.1-
1, imageID=docker-pullable://harbor-001.jimmysong.io/library/spa
rk-driver@sha256:beb92a3e3f178e286d9e5baebdead88b5ba76d651f347ad
2864bb6f8eda26f94, lastState=ContainerState(running=null, termin
ated=null, waiting=null, additionalProperties={}), name=spark-ku
bernetes-driver, ready=true, restartCount=0, state=ContainerState
(running=ContainerStateRunning(startedAt=2017-09-14T06:59:02Z, a
dditionalProperties={}), terminated=null, waiting=null, addition
alProperties={}), additionalProperties={})]
2017-09-14 14:59:12 INFO  LoggingPodStatusWatcherImpl:54 - State
 changed, new state:
     pod name: spark-pi-1505372339796-driver
     namespace: spark-cluster
     labels: spark-app-selector -> spark-f4d3a5d3ad964a05a51feb6
191d50357, spark-role -> driver
     pod uid: 304cf440-991a-11e7-970c-f4e9d49f8ed0
     creation time: 2017-09-14T06:59:01Z
     service account name: spark
     volumes: spark-token-zr8wv
     node name: 172.20.0.114
     start time: 2017-09-14T06:59:01Z
     container images: harbor-001.jimmysong.io/library/spark-dri
ver:v2.1.0-kubernetes-0.3.1-1
     phase: Succeeded
     status: [ContainerStatus(containerID=docker://5c5c821c482a1
e35552adccb567020532b79244392374f25754f0050e6cd4c62, image=harbo
```

```
r-001.jimmysong.io/library/spark-driver:v2.1.0-kubernetes-0.3.1-
1, imageID=docker-pullable://harbor-001.jimmysong.io/library/spa
rk-driver@sha256:beb92a3e3f178e286d9e5baebdead88b5ba76d651f347ad
2864bb6f8eda26f94, lastState=ContainerState(running=null, termin
ated=null, waiting=null, additionalProperties={}), name=spark-ku
bernetes-driver, ready=false, restartCount=0, state=ContainerSta
te(running=null, terminated=ContainerStateTerminated(containerID=
docker://5c5c821c482a1e35552adccb567020532b79244392374f25754f005
0e6cd4c62, exitCode=0, finishedAt=2017-09-14T06:59:11Z, message=
null, reason=Completed, signal=null, startedAt=null, additionalP
roperties={}), waiting=null, additionalProperties={}), additiona
lProperties={})]
2017-09-14 14:59:12 INFO  LoggingPodStatusWatcherImpl:54 - Conta
iner final statuses:


     Container name: spark-kubernetes-driver
     Container image: harbor-001.jimmysong.io/library/spark-driv
er:v2.1.0-kubernetes-0.3.1-1
     Container state: Terminated
     Exit code: 0
2017-09-14 14:59:12 INFO  Client:54 - Application spark-pi finis
hed.
```

从日志中可以看到任务运行的状态信息。

使用下面的命令可以看到 kubernetes 启动的 Pod 信息：

```
kubectl --namespace spark-cluster get pods -w
```

将会看到 `spark-driver` 和 `spark-exec` 的 Pod 信息。

依赖管理

上文中我们在运行测试程序时，命令行中指定的 jar 文件已包含在 docker 镜像中，是不是说我们每次提交任务都需要重新创建一个镜像呢？非也！如果真是这样也太麻烦了。

创建 resource staging server

为了方便用户提交任务，不需要每次提交任务的时候都创建一个镜像，我们使用了 **resource staging server** 。

```
kubectl create -f conf/kubernetes-resource-staging-server.yaml
```

我们同样将其部署在 `spark-cluster` namespace 下，该 yaml 文件见 kubernetes-handbook 的 `manifests/spark-with-kubernetes-native-scheduler` 目录。

优化

其中有一点需要优化，在使用下面的命令提交任务时，使用 `--conf spark.kubernetes.resourceStagingServer.uri` 参数指定 *resource staging server* 地址，用户不应该关注 *resource staging server* 究竟运行在哪台宿主机上，可以使用下面两种方式实现：

- 使用 `nodeSelector` 将 *resource staging server* 固定调度到某一台机器上，该地址依然使用宿主机的 IP 地址
- 改变 `spark-resource-staging-service` service 的 type 为 **ClusterIP**，然后使用 **Ingress** 将其暴露到集群外部，然后加入的内网 DNS 里，用户使用 DNS 名称指定 *resource staging server* 的地址。

然后可以执行下面的命令来提交本地的 jar 到 kubernetes 上运行。

```
./spark-submit \
  --deploy-mode cluster \
  --class org.apache.spark.examples.SparkPi \
  --master k8s://https://172.20.0.113:6443 \
  --kubernetes-namespace spark-cluster \
  --conf spark.kubernetes.authenticate.driver.serviceAccountName=
spark \
  --conf spark.executor.instances=5 \
  --conf spark.app.name=spark-pi \
  --conf spark.kubernetes.driver.docker.image=harbor-001.jimmyso
ng.io/library/spark-driver:v2.1.0-kubernetes-0.3.1-1 \
  --conf spark.kubernetes.executor.docker.image=harbor-001.jimmy
song.io/library/spark-executor:v2.1.0-kubernetes-0.3.1-1 \
  --conf spark.kubernetes.initcontainer.docker.image=harbor-001.
jimmysong.io/library/spark-init:v2.1.0-kubernetes-0.3.1-1 \
  --conf spark.kubernetes.resourceStagingServer.uri=http://172.2
0.0.114:31000 \
  ../examples/jars/spark-examples_2.11-2.2.0-k8s-0.4.0-SNAPSHOT.
jar
```

该命令将提交本地的 `../examples/jars/spark-examples_2.11-2.2.0-k8s-0.4.0-SNAPSHOT.jar` 文件到 *resource staging server*，executor 将从该 server 上获取 jar 包并运行，这样用户就不需要每次提交任务都编译一个镜像了。

详见：https://apache-spark-on-k8s.github.io/userdocs/running-on-kubernetes.html#dependency-management

设置 HDFS 用户

如果 Hadoop 集群没有设置 kerbros 安全认证的话，在指定 `spark-submit` 的时候可以通过指定如下四个环境变量， 设置 Spark 与 HDFS 通信使用的用户：

```
  --conf spark.kubernetes.driverEnv.SPARK_USER=hadoop
  --conf spark.kubernetes.driverEnv.HADOOP_USER_NAME=hadoop
  --conf spark.executorEnv.HADOOP_USER_NAME=hadoop
  --conf spark.executorEnv.SPARK_USER=hadoop
```

使用 hadoop 用户提交本地 jar 包的命令示例：

```
./spark-submit \
  --deploy-mode cluster \
  --class com.talkingdata.alluxio.hadooptest \
  --master k8s://https://172.20.0.113:6443 \
  --kubernetes-namespace spark-cluster \
  --conf spark.kubernetes.driverEnv.SPARK_USER=hadoop \
  --conf spark.kubernetes.driverEnv.HADOOP_USER_NAME=hadoop \
  --conf spark.executorEnv.HADOOP_USER_NAME=hadoop \
  --conf spark.executorEnv.SPARK_USER=hadoop \
  --conf spark.kubernetes.authenticate.driver.serviceAccountName=
spark \
  --conf spark.executor.instances=5 \
  --conf spark.app.name=spark-pi \
  --conf spark.kubernetes.driver.docker.image=harbor-001.jimmyso
ng.io/library/spark-driver:v2.1.0-kubernetes-0.3.1-1 \
  --conf spark.kubernetes.executor.docker.image=harbor-001.jimmy
song.io/library/spark-executor:v2.1.0-kubernetes-0.3.1-1 \
  --conf spark.kubernetes.initcontainer.docker.image=harbor-001.
jimmysong.io/library/spark-init:v2.1.0-kubernetes-0.3.1-1 \
  --conf spark.kubernetes.resourceStagingServer.uri=http://172.2
0.0.114:31000 \
~/Downloads/tendcloud_2.10-1.0.jar
```

详见：https://github.com/apache-spark-on-k8s/spark/issues/408

限制 Driver 和 Executor 的资源使用

在执行 `spark-submit` 时使用如下参数设置内存和 CPU 资源限制：

```
--conf spark.driver.memory=3G
--conf spark.executor.memory=3G
--conf spark.driver.cores=2
--conf spark.executor.cores=10
```

这几个参数中值如何传递到 Pod 的资源设置中的呢？

比如我们设置在执行 `spark-submit` 的时候传递了这样的两个参数：`--conf spark.driver.cores=2` 和 `--conf spark.driver.memory=100G` 那么查看 driver pod 的 yaml 输出结果将会看到这样的资源设置：

```yaml
    resources:
      limits:
        memory: 110Gi
      requests:
        cpu: "2"
        memory: 100Gi
```

以上参数是对 `request` 值的设置，那么 `limit` 的资源设置的值又是从何而来？

可以使用 `spark.kubernetes.driver.limit.cores` 和 `spark.kubernetes.executor.limit.cores` 来设置 CPU的 hard limit。

memory limit 的值是根据 memory request 的值加上 `spark.kubernetes.executor.memoryOverhead` 的值计算而来的，该配置项用于设置分配给每个 executor 的超过 heap 内存的值（可以使用k、m、g单位）。该值用于虚拟机的开销、其他本地服务开销。根据 executor 的大小设置（通常是 6% 到10%）。

我们可以这样来提交一个任务，同时设置 driver 和 executor 的 CPU、内存的资源 request 和 limit 值（driver 的内存 limit 值为 request 值的 110%）。

```
./spark-submit \
  --deploy-mode cluster \
  --class org.apache.spark.examples.SparkPi \
  --master k8s://https://172.20.0.113:6443 \
  --kubernetes-namespace spark-cluster \
  --conf spark.kubernetes.authenticate.driver.serviceAccountName=
spark \
  --conf spark.driver.memory=100G \
  --conf spark.executor.memory=10G \
  --conf spark.driver.cores=30 \
  --conf spark.executor.cores=2 \
  --conf spark.driver.maxResultSize=10240m \
  --conf spark.kubernetes.driver.limit.cores=32 \
  --conf spark.kubernetes.executor.limit.cores=3 \
  --conf spark.kubernetes.executor.memoryOverhead=2g \
  --conf spark.executor.instances=5 \
  --conf spark.app.name=spark-pi \
  --conf spark.kubernetes.driver.docker.image=harbor-001.jimmyso
ng.io/library/spark-driver:v2.1.0-kubernetes-0.3.1-1 \
  --conf spark.kubernetes.executor.docker.image=harbor-001.jimmy
song.io/library/spark-executor:v2.1.0-kubernetes-0.3.1-1 \
  --conf spark.kubernetes.initcontainer.docker.image=harbor-001.
jimmysong.io/library/spark-init:v2.1.0-kubernetes-0.3.1-1 \
local:///opt/spark/examples/jars/spark-examples_2.11-2.2.0-k8s-0
.4.0-SNAPSHOT.jar 10000000
```

这将启动一个包含一千万个 task 的计算 pi 的 spark 任务，任务运行过程中，drvier 的 CPU 实际消耗大约为 3 核，内存 40G，每个 executor 的 CPU 实际消耗大约不到 1 核，内存不到 4G，我们可以根据实际资源消耗不断优化资源的 request 值。

`SPARK_DRIVER_MEMORY` 和 `SPARK_EXECUTOR_MEMORY` 和分别作为 Driver 容器和 Executor 容器启动的环境变量，比如下面这个 Driver 启动的 CMD 中：

```
CMD SPARK_CLASSPATH="${SPARK_HOME}/jars/*" && \
    env | grep SPARK_JAVA_OPT_ | sed 's/[^=]*=\(.*\)/\1/g' > /tmp/java_opts.txt && \
    readarray -t SPARK_DRIVER_JAVA_OPTS < /tmp/java_opts.txt && \
    if ! [ -z ${SPARK_MOUNTED_CLASSPATH+x} ]; then SPARK_CLASSPATH="$SPARK_MOUNTED_CLASSPATH:$SPARK_CLASSPATH"; fi && \
    if ! [ -z ${SPARK_SUBMIT_EXTRA_CLASSPATH+x} ]; then SPARK_CLASSPATH="$SPARK_SUBMIT_EXTRA_CLASSPATH:$SPARK_CLASSPATH"; fi && \
    if ! [ -z ${SPARK_EXTRA_CLASSPATH+x} ]; then SPARK_CLASSPATH="$SPARK_EXTRA_CLASSPATH:$SPARK_CLASSPATH"; fi && \
    if ! [ -z ${SPARK_MOUNTED_FILES_DIR+x} ]; then cp -R "$SPARK_MOUNTED_FILES_DIR/." .; fi && \
    if ! [ -z ${SPARK_MOUNTED_FILES_FROM_SECRET_DIR} ]; then cp -R "$SPARK_MOUNTED_FILES_FROM_SECRET_DIR/." .; fi && \
    ${JAVA_HOME}/bin/java "${SPARK_DRIVER_JAVA_OPTS[@]}" -cp $SPARK_CLASSPATH -Xms$SPARK_DRIVER_MEMORY -Xmx$SPARK_DRIVER_MEMORY $SPARK_DRIVER_CLASS $SPARK_DRIVER_ARGS
```

我们可以看到对 `SPARK_DRIVER_MEMORY` 环境变量的引用。Executor 的设置与 driver 类似。

而我们可以使用这样的参数来传递环境变量的值 `spark.executorEnv.[EnvironmentVariableName]`，只要将 `EnvironmentVariableName` 替换为环境变量名称即可。

参考

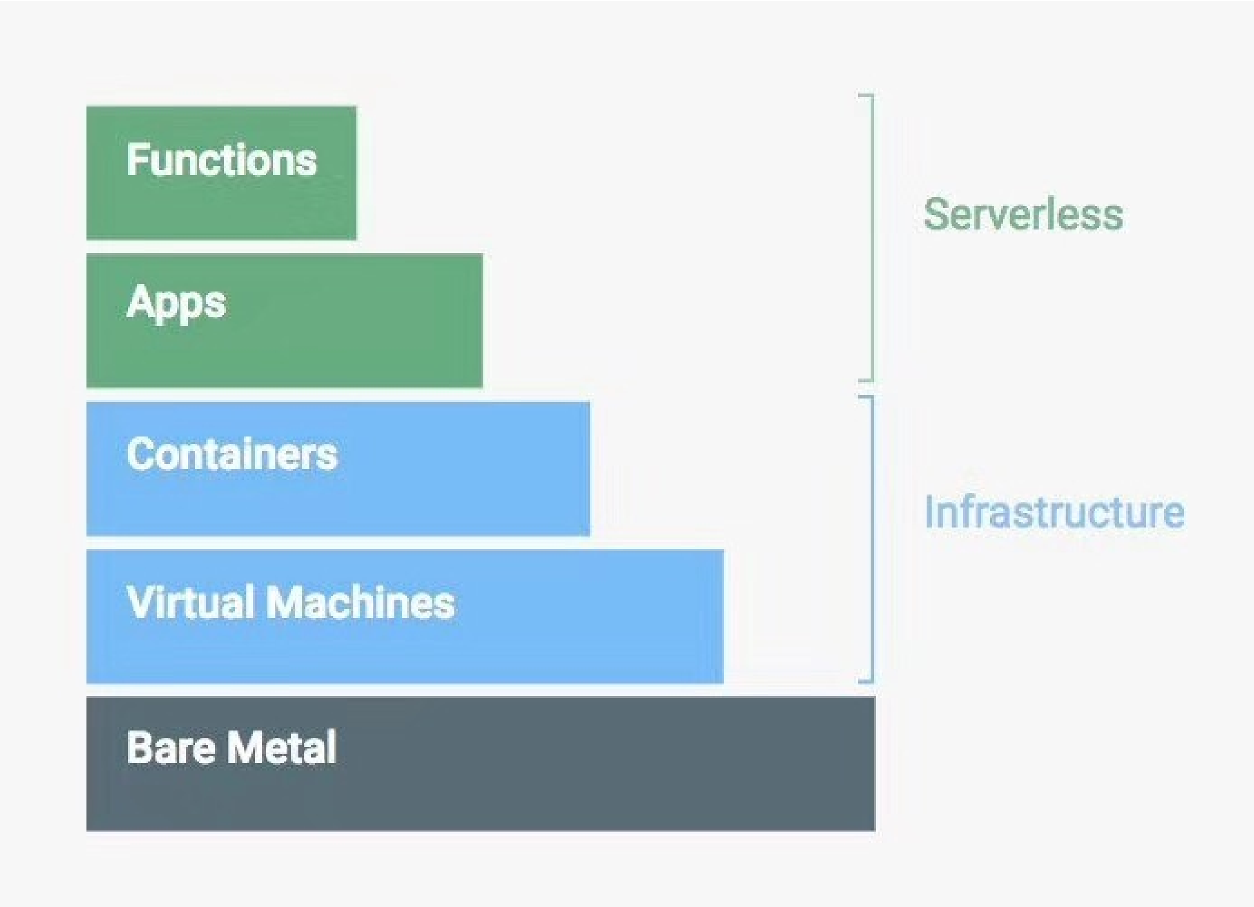
- Spark动态资源分配-Dynamic Resource Allocation
- Running Spark on Kubernetes
- Apache Spark Jira Issue - 18278 - SPIP: Support native submission of spark jobs to a kubernetes cluster
- Kubernetes Github Issue - 34377 Support Spark natively in Kubernetes
- Kubernetes example spark
- https://github.com/rootsongjc/spark-on-kubernetes
- Scheduler backend
- Introduction to Spark on Kubernetes - banzaicloud.com
- Scaling Spark made simple on Kubernetes - banzaicloud.com
- The anatomy of Spark applications on Kubernetes - banzaicloud.com
- Monitoring Apache Spark with Prometheus - banzaicloud.com

- Running Zeppelin Spark notebooks on Kubernetes - banzaicloud.com
- Apache Spark CI/CD workflow howto - banzaicloud.com

Serverless架构

Serverless（无服务器架构）指的是由开发者实现的服务端逻辑运行在无状态的计算容器中，它由事件触发， 完全被第三方管理，其业务层面的状态则被开发者使用的数据库和存储资源所记录。

下图来自谷歌云平台官网，是对云计算的一个很好的分层概括，其中 serverless 就是构建在虚拟机和容器之上的一层，与应用本身的关系更加密切。



图片 - 从物理机到函数计算

Serverless架构的优点

今天大多数公司在开发应用程序并将其部署在服务器上的时候，无论是选择公有云还是私有的数据中心，都需要提前了解究竟需要多少台服务器、多大容量的存储和数据库的功能等。并需要部署运行应用程序和依赖的软件到基础设施之上。假设我

们不想在这些细节上花费精力，是否有一种简单的架构模型能够满足我们这种想法？这个答案已经存在，这就是今天软件架构世界中新鲜但是很热门的一个话题——Serverless（无服务器）架构。

——AWS 费良宏

- 降低运营成本：

Serverless是非常简单的外包解决方案。它可以让您委托服务提供商管理服务器、数据库和应用程序甚至逻辑，否则您就不得不自己来维护。由于这个服务使用者的数量会非常庞大，于是就会产生规模经济效应。在降低成本上包含了两个方面，即基础设施的成本和人员（运营/开发）的成本。

- 降低开发成本：

IaaS和PaaS存在的前提是，服务器和操作系统管理可以商品化。Serverless作为另一种服务的结果是整个应用程序组件被商品化。

- 扩展能力：

Serverless架构一个显而易见的优点即“横向扩展是完全自动的、有弹性的、且由服务提供者所管理”。从基本的基础设施方面受益最大的好处是，您只需支付您所需要的计算能力。

- 更简单的管理：

Serverless架构明显比其他架构更简单。更少的组件，就意味着您的管理开销会更少。

- “绿色”的计算：

按照《福布斯》杂志的统计，在商业和企业数据中心的典型服务器仅提供5%～15%的平均最大处理能力的输出。这无疑是一种资源的巨大浪费。随着Serverless架构的出现，让服务提供商提供我们的计算能力最大限度满足实时需求。这将使我们更有效地利用计算资源。

Kubernetes上的serverless 架构

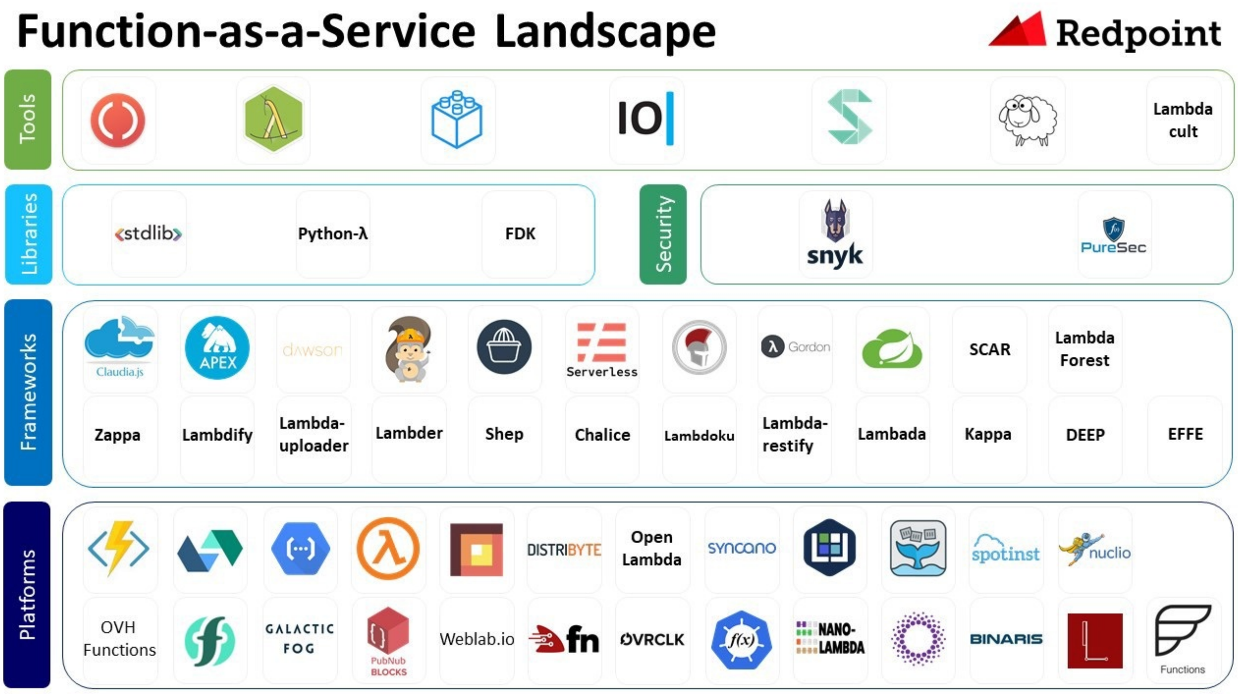
目前已经有一批优秀的基于 kubernetes 的 serverless 架构（FaaS）开源项目如下：

- faas - Functions as a Service - a serverless framework for Docker & Kubernetes https://blog.alexellis.io/introducing…
- faas-netes - Enable Kubernetes as a backend for Functions as a Service (OpenFaaS) https://github.com/alexellis/faas
- fn - The container native, cloud agnostic serverless platform. http://fnproject.io
- funktion - a CLI tool for working with funktion https://funktion.fabric8.io/
- fx - Poor man's serverless framework based on Docker, Function as a Service with painless.
- IronFunctions - IronFunctions - the serverless microservices platform. http://iron.io
- kubeless - Kubernetes Native Serverless Framework http://kubeless.io
- OpenWhisk - Apache OpenWhisk (Incubating) is a serverless, open source cloud platform that executes functions in response to events at any scale.

以上项目收录于 awsome-cloud-native

FaaS

Function-as-a-Service 全景图（图片来自 https://github.com/amyers1793/FunctionasaServiceLandscape)



图片 - *FaaS Landscape*

参考

Serverless Architectures - Martin Fowler

Serverless架构综述

2017年会是Serverless爆发之年吗？

从IaaS到FaaS—— Serverless架构的前世今生

Introducing Redpoint's FaaS Landscape

Updated at 2018-05-09 14:15:47

理解Serverless

No silver bullet. - The Mythical Man-Month

许多年前，我们开发的软件还是C/S（客户端/服务器）和MVC（模型-试图-控制器）的形式，再后来有了SOA，最近几年又出现了微服务架构，更新一点的有Cloud Native（云原生）应用，企业应用从单体架构，到服务化，再到更细粒度的微服务化，应用开发之初就是为了应对互联网的特有的高并发、不间断的特性，需要很高的性能和可扩展性，人们对软件开发的追求孜孜不倦，希望力求在软件开发的复杂度和效率之间达到一个平衡。但可惜的是，NO SILVER BULLET！几十年前（1975年）Fred Brooks就在The Mythical Man-Month中就写到了这句话。那么Serverlss会是那颗银弹吗？

云改变了我们对操作系统的认知，原来一个系统的计算资源、存储和网络是可以分离配置的，而且还可以弹性扩展，但是长久以来，我们在开发应用时始终没有摆脱的服务器的束缚（或者说认知），应用必须运行在不论是实体还是虚拟的服务器上，必须经过部署、配置、初始化才可以运行，还需要对服务器和应用进行监控和管理，还需要保证数据的安全性，这些云能够帮我们简化吗？让我们只要关注自己代码的逻辑就好了，其它的东西让云帮我实现就好了。

Serverless介绍

Serverless架构是云的自然延伸，为了理解serverless，我们有必要回顾一下云计算的发展。

IaaS

2006年AWS推出EC2（Elastic Compute Cloud），作为第一代IaaS（Infrastructure as a Service），用户可以通过AWS快速的申请到计算资源，并在上面部署自己的互联网服务。IaaS从本质上讲是服务器租赁并提供基础设施外包服务。就比如我们用的水和电一样，我们不会自己去引入自来水和发电，而是直接从自来水公司和电网公司购入，并根据实际使用付费。

EC2真正对IT的改变是硬件的虚拟化（更细粒度的虚拟化），而EC2给用户带来了以下五个好处：

- 降低劳动力成本：减少了企业本身雇佣IT人员的成本
- 降低风险：不用再像自己运维物理机那样，担心各种意外风险，EC2有主机损坏，再申请一个就好了。
- 降低基础设施成本：可以按小时、周、月或者年为周期租用EC2。
- 扩展性：不必过早的预期基础设施采购，因为通过云厂商可以很快的获取。
- 节约时间成本：快速的获取资源开展业务实验。

以上说了是IaaS或者说基础设施外包的好处，当然其中也有弊端，我们将在后面讨论。

以上是AWS为代表的公有云IaaS，还有使用OpenStack构建的私有云也能够提供IaaS能力。

PaaS

PaaS（Platform as a Service）是构建在IaaS之上的一种平台服务，提供操作系统安装、监控和服务发现等功能，用户只需要部署自己的应用即可，最早的一代是Heroku。Heroko是商业的PaaS，还有一个开源的PaaS——Cloud Foundry，用户可以基于它来构建私有PaaS，如果同时使用公有云和私有云，如果能在两者之间构建一个统一的PaaS，那就是“混合云”了。

在PaaS上最广泛使用的技术就要数docker了，因为使用容器可以很清晰的描述应用程序，并保证环境一致性。管理云上的容器，可以称为是CaaS（Container as a Service），如GCE（Google Container Engine）。也可以基于Kubernetes、Mesos这类开源软件构件自己的CaaS，不论是直接在IaaS构建还是基于PaaS。

PaaS是对软件的一个更高的抽象层次，已经接触到应用程序的运行环境本身，可以由开发者自定义，而不必接触更底层的操作系统。

Serverless的定义

Serverless不如IaaS和PaaS那么好理解，因为它通常包含了两个领域BaaS（Backend as a Service）和FaaS（Function as a Service）。

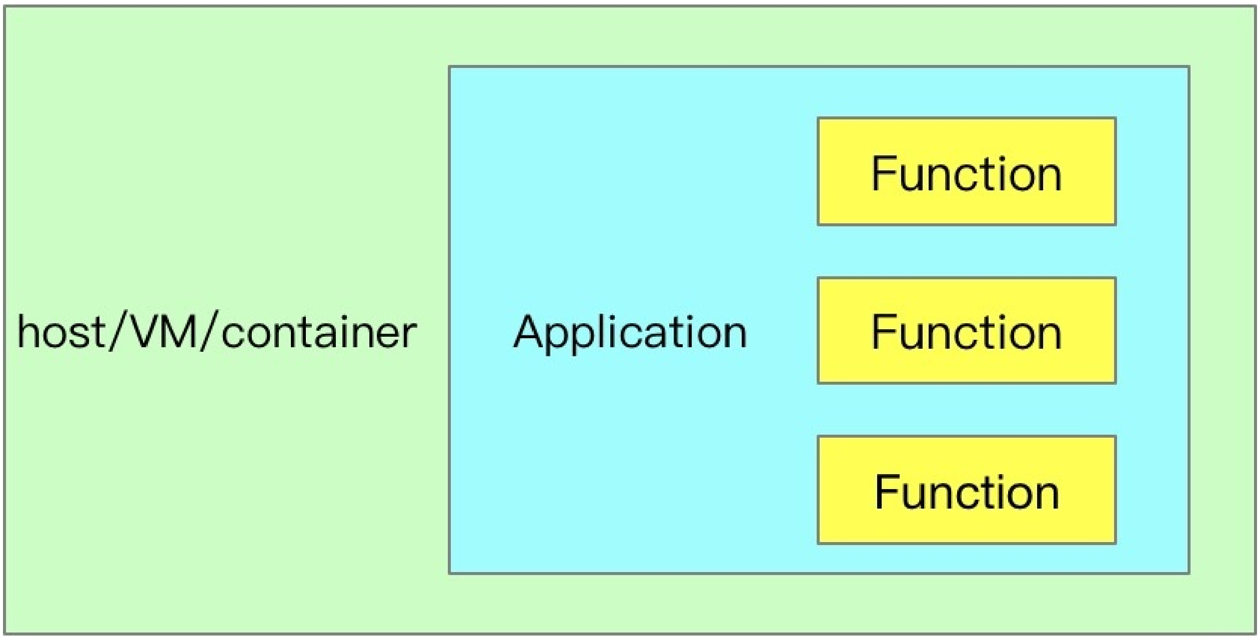
BaaS

BaaS（Backend as a Service）后端即服务，一般是一个个的API调用后端或别人已经实现好的程序逻辑，比如身份验证服务Auth0，这些BaaS通常会用来管理数据，还有很多公有云上提供的我们常用的开源软件的商用服务，比如亚马逊的RDS可以替代我们自己部署的MySQL，还有各种其它数据库和存储服务。

FaaS

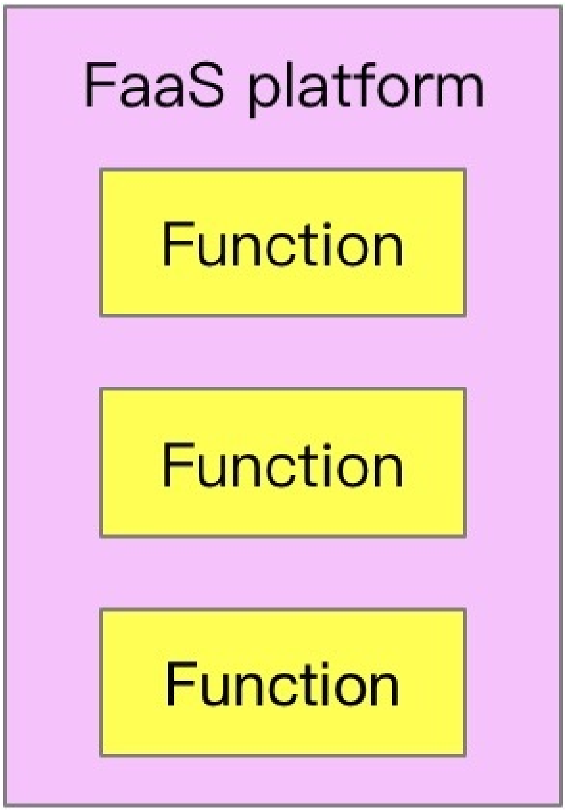
FaaS（Functions as a Service）函数即服务，FaaS是无服务器计算的一种形式，当前使用最广泛的是AWS的Lambada。

现在当大家讨论Serverless的时候首先想到的就是FaaS，有点甚嚣尘上了。FaaS本质上是一种事件驱动的由消息触发的服务，FaaS供应商一般会集成各种同步和异步的事件源，通过订阅这些事件源，可以突发或者定期的触发函数运行。



图片 - 服务端软件的运行环境

传统的服务器端软件不同是经应用程序部署到拥有操作系统的虚拟机或者容器中，一般需要长时间驻留在操作系统中运行，而FaaS是直接将程序部署上到平台上即可，当有事件到来时触发执行，执行完了就可以卸载掉。



图片 - *FaaS*应用架构

总结

两者都为我们的计算资源提供了弹性的保障，BaaS其实依然是服务外包，而FaaS使我们更加关注应用程序的逻辑，两者使我们不需要关注应用程序所在的服务器，但实际上服务器依然是客观存在的。

当我们将应用程序迁移到容器和虚拟机中时，其实对于应用程序本身的体系结构并没有多少改变，只不过有些流程和规定需要遵守，比如12因素应用守则，但是serverlss对应用程序的体系结构来说就是一次颠覆了，通常我们需要考虑事件驱动模型，更加细化的不熟形式，以及在FaaS组件之外保持状态的需求。

Serverless应用

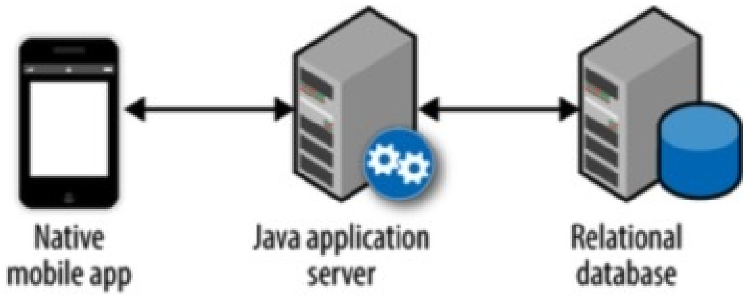
我们以一个游戏应用为例，来说明什么是serverless应用。

一款移动端游戏至少包含如下几个特性：

- 移动端友好的用户体验

- 用户管理和权限认证
- 关卡、升级等游戏逻辑，游戏排行，玩家的等级、任务等信息

传统的应用程序架构可能是这样的：

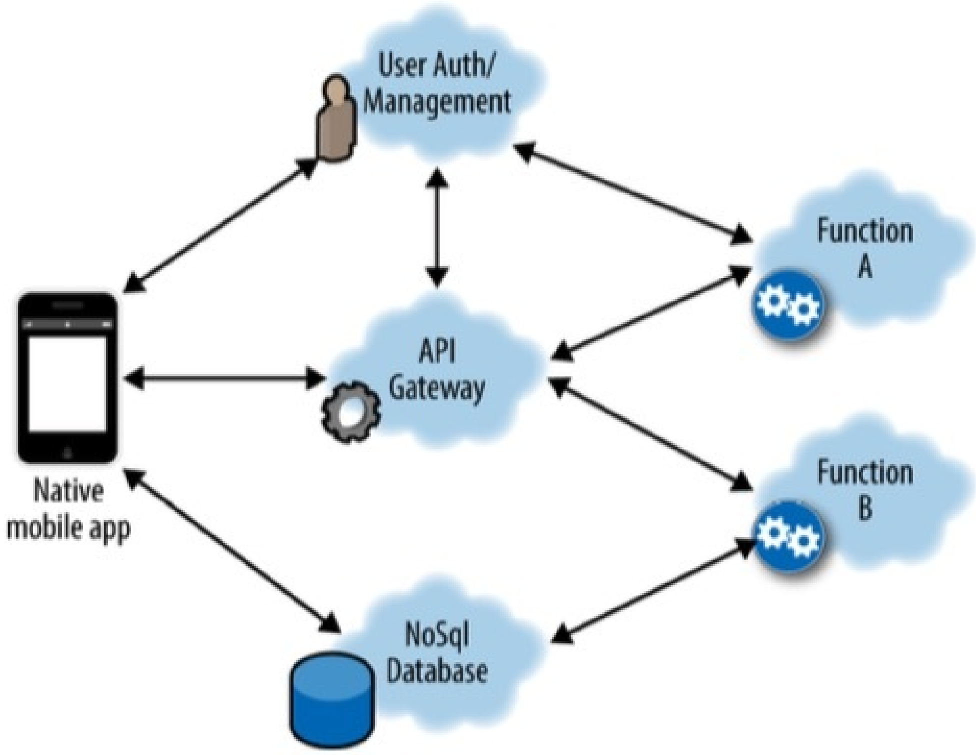


图片 - 传统应用程序架构

- 一个app前端，iOS后者安卓
- 用Java写的后端，使用JBoss或者Tomcat做server运行
- 使用关系型数据库存储用户数据，如MySQL

这样的架构可以让前端十分轻便，不需要做什么应用逻辑，只是负责渲染用户界面，将请求通过HTTP发送给后端，而所有的数据操作都是有由后端的Java程序来完成的。

这样的架构开发起来比较容易，但是维护起来确十分复杂，前端开发、后端的开发都需要十分专业的人员、环境的配置，还要有人专门维护数据库、应用的更新和升级。



图片 - *Serverless*架构

而在serverless架构中，我们不再需要在服务器端代码中存储任何会话状态，而是直接将它们存储在NoSQL中，这样将使应用程序无状态，有助于弹性扩展。前端可以直接利用BaaS而减少后端的编码需求，这样架构的本质上是减少了应用程序开发的人力成本，降低了自己维护基础设施的风险，而且利用云的能力更便于扩展和快速迭代。

Serverless的优势

在最前面我们提到了使用IaaS给我们带来了五点好处，FaaS当然也包括了这些好处，但是它给我们带来的最大的好处就是多快好省。减少从概念原型到实施的等待时间，比自己维护服务更省钱。

降低人力成本

不需要再自己维护服务器，操心服务器的各种性能指标和资源利用率，而是关心应用程序本身的状态和逻辑。而且serverless应用本身的部署也十分容易，我们只要上传基本的代码但愿，例如Javascript或Python的源代码的zip文件，以及基于JVM的语言的纯JAR文件。不需使用Puppet、Chef、Ansible或Docker来进行配置管

理，降低了运维成本。同时，对于运维来说，也不再需要监控那些更底层的如磁盘使用量、CPU使用率等底层和长期的指标信息，而是监控应用程序本身的度量，这将更加直观和有效。

在此看来有人可能会提出“NoOps”的说法，其实这是不存在的，只要有应用存在的一天就会有Ops，只是人员的角色会有所转变，部署将变得更加自动化，监控将更加面向应用程序本身，更底层的运维依然需要专业的人员去做。

降低风险

对于组件越多越复杂的系统，出故障的风险就越大。我们使用BaaS或FaaS将它们外包出去，让专业人员来处理这些故障，有时候比我们自己来修复更可靠，利用专业人员的知识来降低停机的风险，缩短故障修复的时间，让我们的系统稳定性更高。

减少资源开销

我们在申请主机资源一般会评估一个峰值最大开销来申请资源，往往导致过度的配置，这意味着即使在主机闲置的状态下也要始终支付峰值容量的开销。对于某些应用来说这是不得已的做法，比如数据库这种很难扩展的应用，而对于普通应用这就显得不太合理了，虽然我们都觉得即使浪费了资源也比当峰值到来时应用程序因为资源不足而挂掉好。

解决这个问题最好的办法就是，不计划到底需要使用多少资源，而是根据实际需要来请求资源，当然前提必须是整个资源池是充足的（公有云显然更适合）。根据使用时间来付费，根据每次申请的计算资源来付费，让计费的粒度更小，将更有利于降低资源的开销。这是对应用程序本身的优化，例如让每次请求耗时更短，让每次消耗的资源更少将能够显著节省成本。

增加缩放的灵活性

以AWS Lamba为例，当平台接收到第一个触发函数的事件时，它将启动一个容器来运行你的代码。如果此时收到了新的事件，而第一个容器仍在处理上一个事件，平台将启动第二个代码实例来处理第二个事件。AWS lambad的这种自动的零管理水平缩放，将持续到有足够的代码实例来处理所有的工作负载。

但是，AWS仍然只会向您收取代码的执行时间，无论它需要启动多少个容器实例要满足你的负载请求。例如，假设所有事件的总执行时间是相同的，在一个容器中按顺序调用Lambda 100次与在100个不同容器中同时调用100次Lambda的成本是 一

样的。当然AWS Lambada也不会无限制的扩展实例个数，如果有人对你发起了DDos攻击怎么办，那么不就会产生高昂的成本吗？AWS是有默认限制的，默认执行Lambada函数最大并发数是1000。

缩短创新周期

小团队的开发人员正可以在几天之内从头开始开发应用程序并部署到生产。使用短而简单的函数和事件来粘合强大的驱动数据存储和服务的API。完成的应用程序具有高度可用性和可扩展性，利用率高，成本低，部署速度快。

以docker为代表的容器技术仅仅是缩短了应用程序的迭代周期，而serverless技术是直接缩短了创新周期，从概念到最小可行性部署的时间，让初级开发人员也能在很短的时间内完成以前通常要经验丰富的工程师才能完成的项目。

Serverless的劣势

我们知道没有十全十美的技术，在说了serverless的那么多优势之后，我们再来探讨以下serverless的劣势，或者说局限性和适用场景。

状态管理

要想实现自由的缩放，无状态是必须的，而对于有状态的服务，使用serverless这就丧失了灵活性，有状态服务需要与存储交互就不可避免的增加了延迟和复杂性。

延迟

应用程序中不同组件的访问延迟是一个大问题，我们可以通过使用专有的网络协议、RPC调用、数据格式来优化，或者是将实例放在同一个机架内或同一个主机实例上来优化以减少延迟。

而serverless应用程序是高度分布式、低耦合的，这就意味着延迟将始终是一个问题，单纯使用serverless的应用程序是不太现实的。

本地测试

Serverless应用的本地测试困难是一个很棘手的问题。虽然可以在测试环境下使用各种数据库和消息队列来模拟生产环境，但是对于无服务应用的集成或者端到端测试尤其困难，很难在本地模拟应用程序的各种连接，并与性能和缩放的特性结合起来测试，并且serverless应用本身也是分布式的，简单的将无数的FaaS和BaaS组件粘合起来也是有挑战性的。

FaaS-函数即服务

FaaS（Functions as a Service）函数即服务，FaaS是无服务器计算的一种形式，当前使用最广泛的是AWS的Lambada。

现在当大家讨论Serverless的时候首先想到的就是FaaS，有点甚嚣尘上了。FaaS本质上是一种事件驱动的由消息触发的服务，FaaS供应商一般会集成各种同步和异步的事件源，通过订阅这些事件源，可以突发或者定期的触发函数运行。

当前开源的FaaS框架大部分都是基于Kuberentes来实现的，例如：

- faas-netes - Enable Kubernetes as a backend for Functions as a Service (OpenFaaS) https://github.com/alexellis/faas
- fn - The container native, cloud agnostic serverless platform. http://fnproject.io
- funktion - a CLI tool for working with funktion https://funktion.fabric8.io/
- fx - Poor man's serverless framework based on Docker, Function as a Service with painless.
- IronFunctions - IronFunctions - the serverless microservices platform. http://iron.io
- kubeless - Kubernetes Native Serverless Framework http://kubeless.io
- nuclio - High-Performance Serverless event and data processing platform
- OpenFaaS - OpenFaaS - Serverless Functions Made Simple for Docker & Kubernetes https://blog.alexellis.io/introducing-functions-as-a-service/
- OpenWhisk - Apache OpenWhisk (Incubating) is a serverless, open source cloud platform that executes functions in response to events at any scale.

关于整个Cloud Native开源生态，请参考awesome-cloud-native。

OpenFaaS快速入门指南

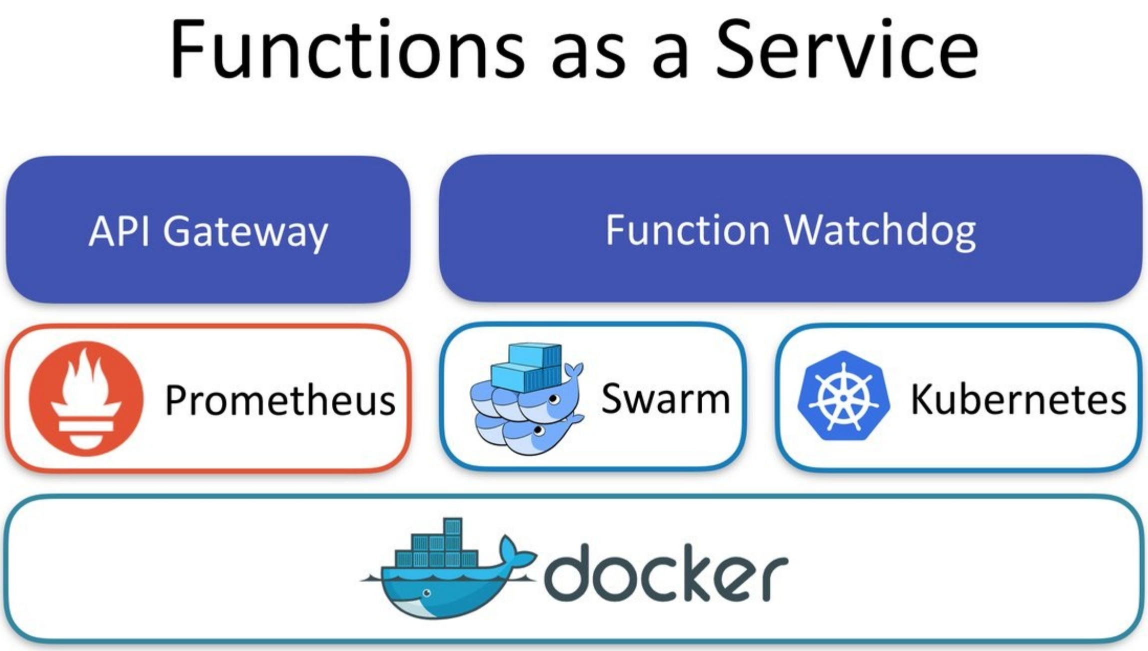
OpenFaaS一款高人气的开源的faas框架，可以直接在Kubernetes上运行，也可以基于Swarm或容器运行。

在Kubernetes上部署OpenFaaS十分简单，用到的镜像如下：

- functions/faas-netesd:0.3.4
- functions/gateway:0.6.14
- functions/prometheus:latest-k8s
- functions/alertmanager:latest-k8s

这些镜像都存储在DockerHub上。

OpenFaaS的架构如下图：



图片 - *OpenFaaS*架构

部署

参考Deployment guide for Kubernetes部署OpenFaaS。

如果您的Kuberentes集群可以访问DockerHub那么直接使用官方提供的YAML文件即可。

YAML文件见官方仓库：https://github.com/openfaas/faas-netes

部署同步请求

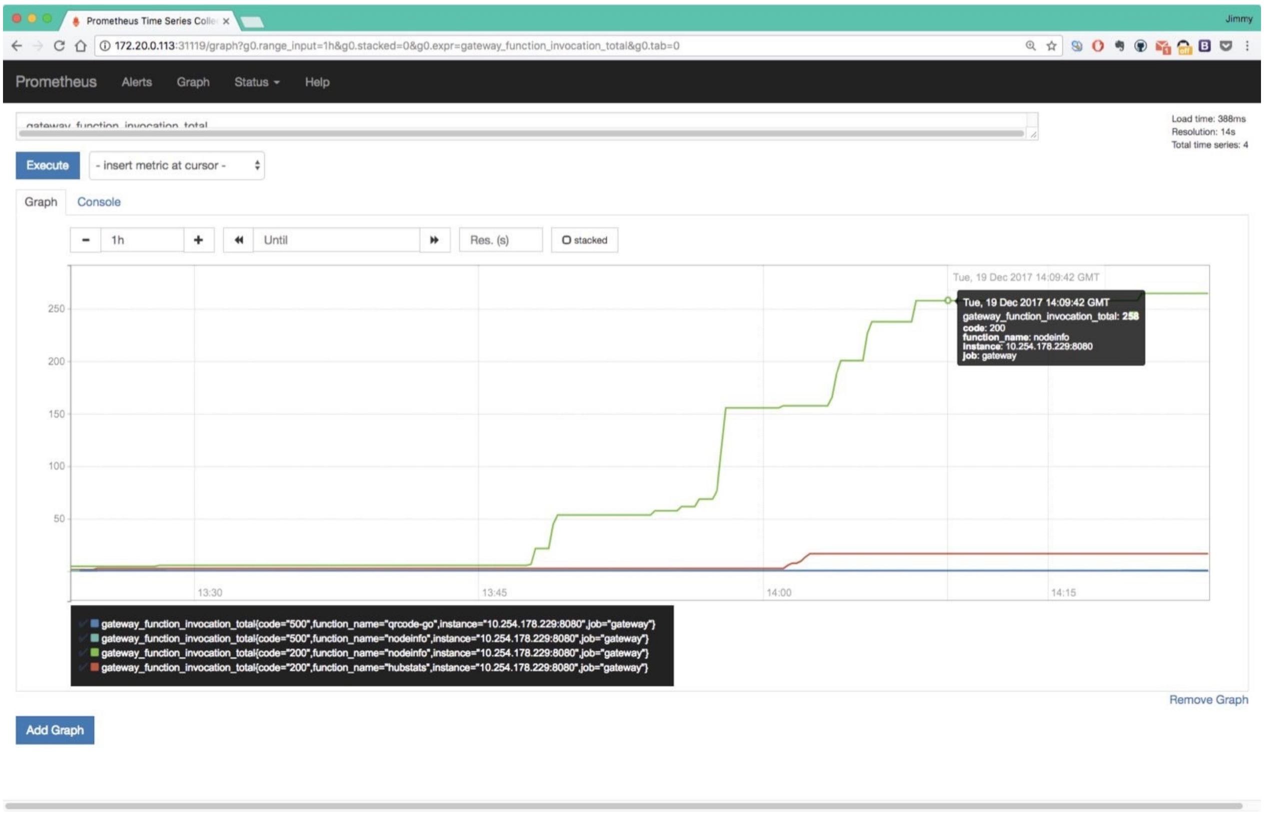
一共用到了三个YAML文件：

- faas.yml
- monitoring.yml
- rbac.yml

访问端口

服务	TCP端口
API Gateway/UI	31112
Prometheus	31119

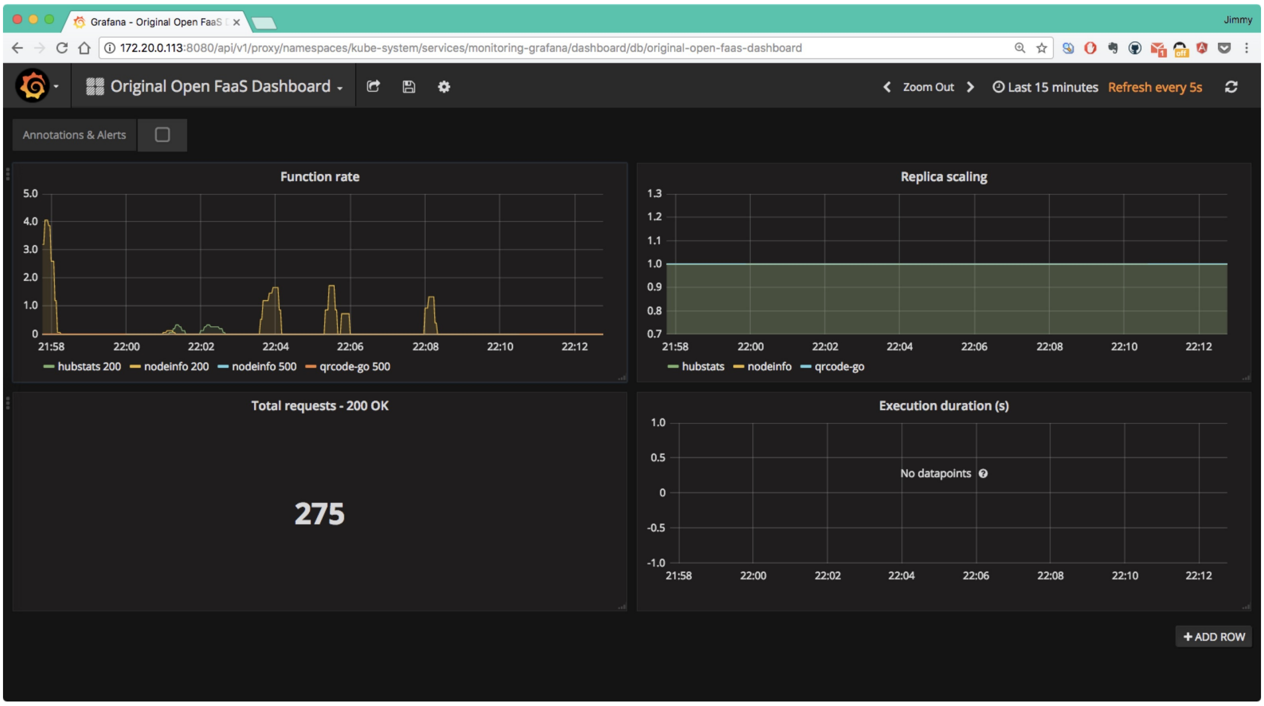
OpenFaaS安装好后会启动一个Prometheus，使用31119端口，通过任意一个node可以访问UI：http://172.20.0.113:31119



图片 - *OpenFaaS Prometheus*

在这里可以看到函数的运行情况。

同时OpenFaaS还提供了Dashboard，需要我们自己向Grafana中配置Prometheus数据源后导入，JSON配置见：https://grafana.com/dashboards/3526，可以下载后直接导入到Grafana中。

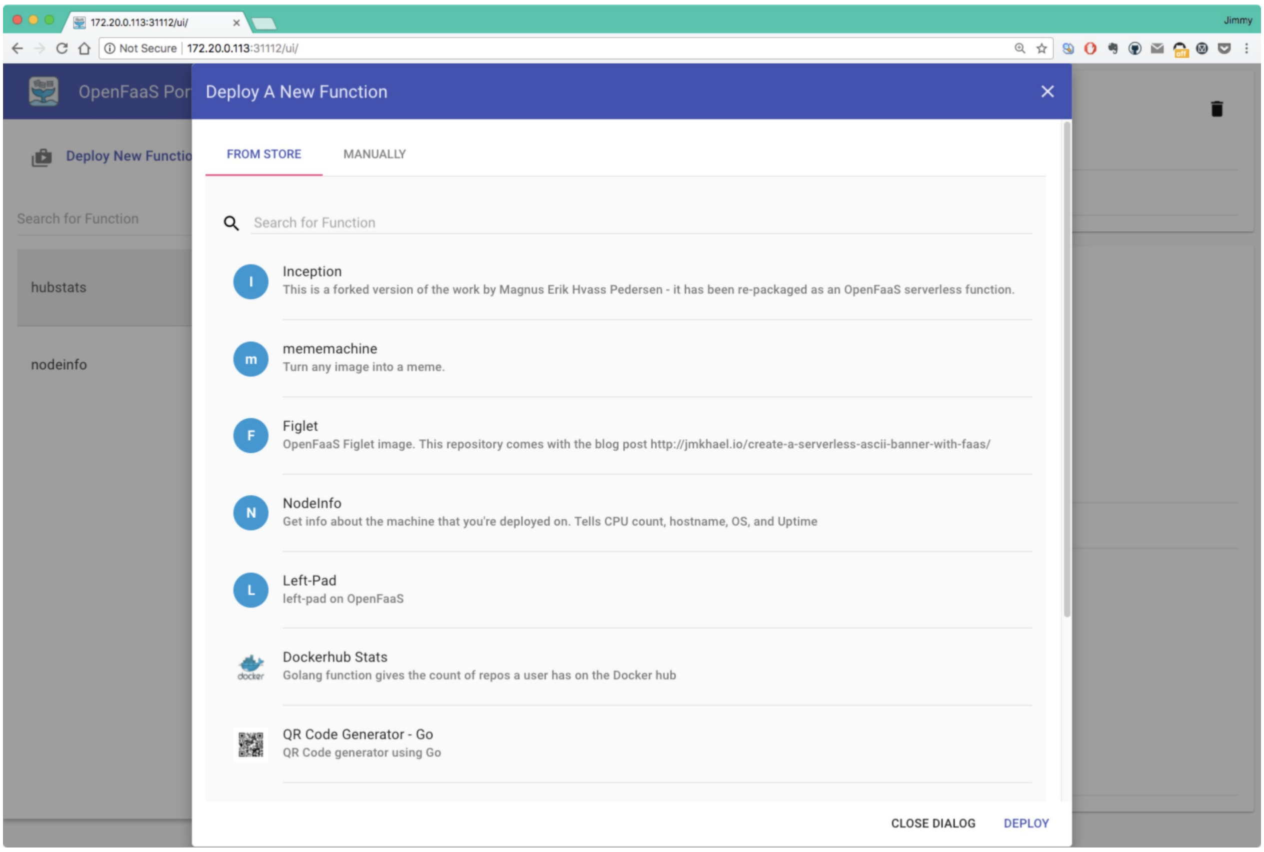


图片 - *OpenFaas Grafana*监控

OpenFaaS的使用

OpenFaaS提供了便捷的UI，在部署完成后就可以通过NodePort方式访问。

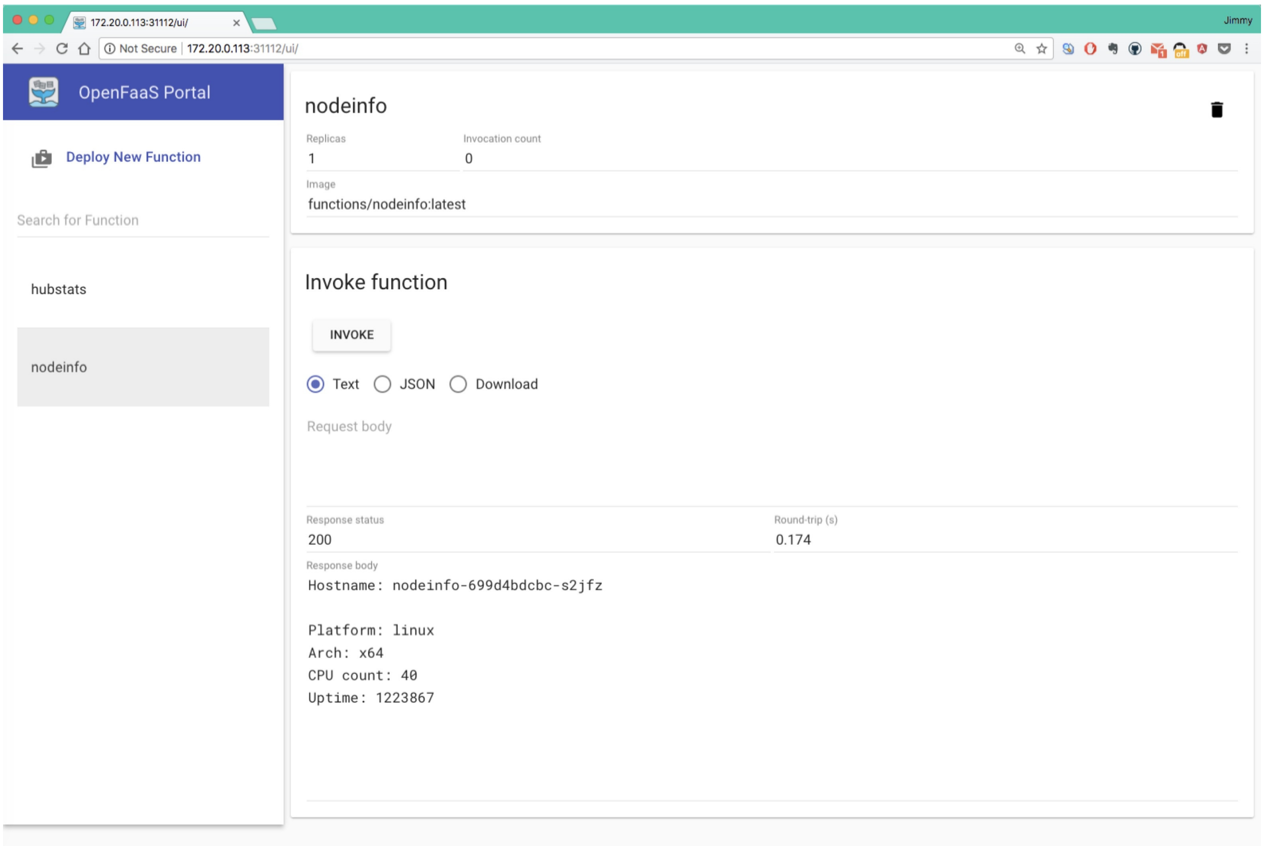
使用API Gateway的端口，通过任意一个node可以访问UI：http://172.20.0.113:31112



图片 - *OpenFaas Dashboard*

其中已经内置了一些函数应用可供我们试用，还可以创建自己的函数。

比如内置的 `NodeInfo` 应用，可以获取该应用所部署到的主机的信息，如下图：



图片 - *NodeInfo*执行结果

注意：有一些js和css文件需要翻墙才能访问，否则页面将出现格式错误。

命令行工具

OpenFaaS提供了命令行工具faas-cli，使用该工具可以管理OpenFaaS中的函数。

可以到openfaas GitHub release下载对应操作系统的命令行工具。或者使用下面的命令安装最新faas-cli：

```
curl -sL cli.openfaas.com | sudo sh
```

faas-cli命令说明

下面是 `faas-cli` 命令的几个使用案例。

获取当前部署的函数状态：

```
faas-cli list --gateway http://172.20.0.113:31112
Function                             Invocations         Replicas
hubstats                             0                   1
nodeinfo                             0                   1
```

调用函数nodeinfo：

```
echo ""|faas-cli invoke nodeinfo --gateway http://172.20.0.113:3
1112
Hostname: nodeinfo-699d4bdcbc-s2jfz

Platform: linux
Arch: x64
CPU count: 40
Uptime: 1728200
```

OpenFaaS的命令行工具 `faas-cli` 的详细使用说明见：https://github.com/openfaas/faas-cli

参考

Deployment guide for Kubernetes - GitHub openfaas/faas

边缘计算

TBD

参考

[The Birth of an Edge Orchestrator – Cloudify Meets Edge Computing](#)

[K8s(Kubernetes) and SDN for Multi-access Edge Computing deployment](#)

人工智能

Kubernetes 在人工智能领域的应用。

TBD

- kubeflow - Kubernetes 机器学习工具箱

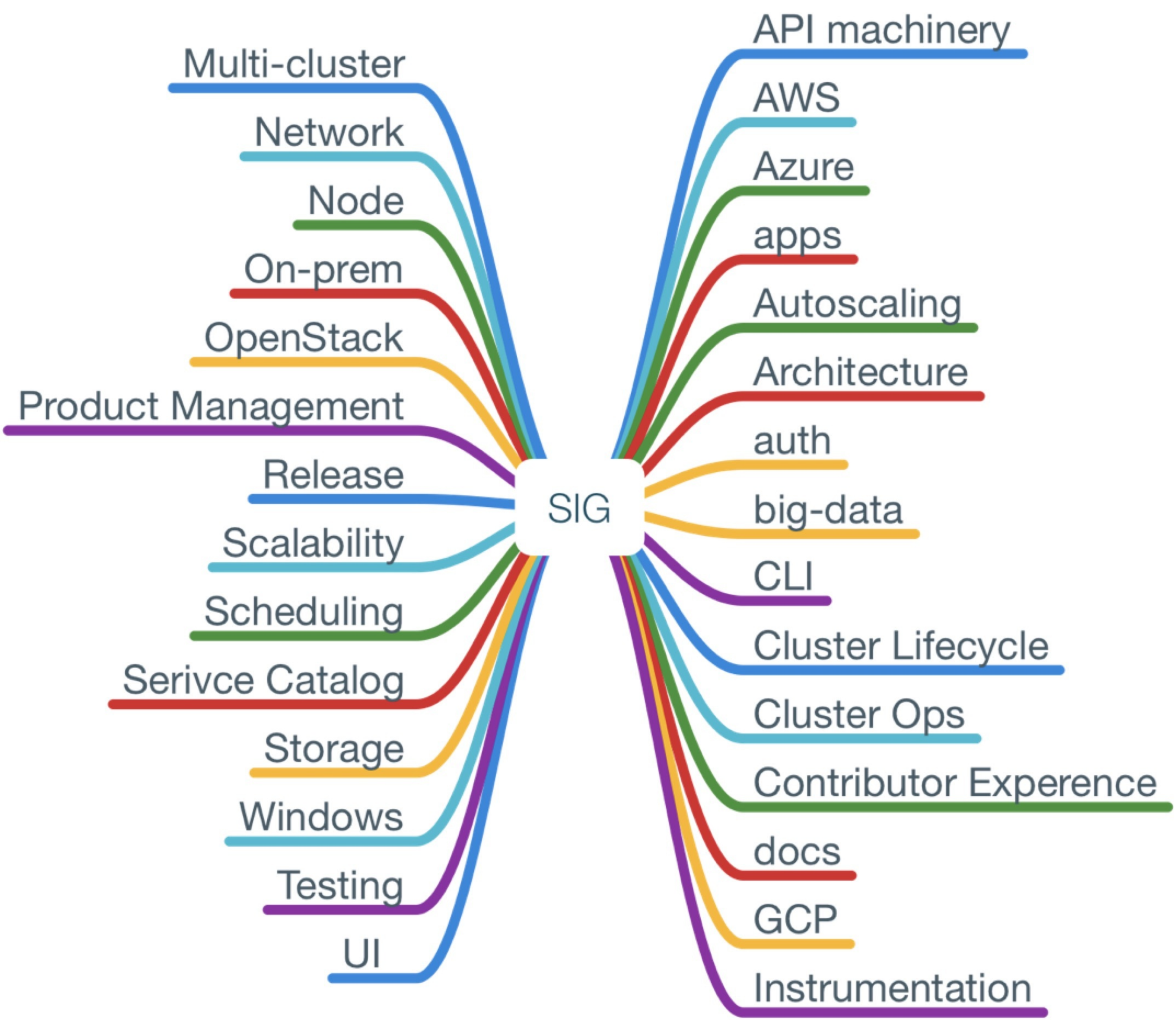
开发指南说明

讲解如何在原生 Kubernetes 的基础上做定制开发。

SIG和工作组

Kubernetes的社区是以SIG（Special Interest Group特别兴趣小组）和工作组的形式组织起来的，每个工作组都会定期召开视频会议。

所有的SIG和工作组都使用slack和邮件列表沟通。



图片 - *Kubernetes SIG*

主要SIG列表

- **api-machinery**：所有API级别的功能，包括了API server、API注册和发现、

通用的API CRUD语义，准入控制，编码/解码，转换，默认值，持久化层（etcd），OpenAPI，第三方资源，垃圾手机和客户端库的方方面面。

- **aws**：如何在AWS上支持和使用kubernetes。
- **apps**：在kubernetes上部署和运维应用程序。关注开发者和DevOps在kubernetes上运行应用程序的体验。
- **architecture**：维持kubernetes在架构设计上的一致性和原则。
- **auth**：kubernetes的认证授权、权限管理和安全性策略。
- **autoscaling**：集群的自动缩放，pod的水平和垂直自动缩放，pod的资源初始化，pod监控和指标收集等主题。
- **azure**：如何在Azure上支持和使用kubernetes。
- **big-data**：在kubernetes上部署和运行大数据应用，如Spark、Kafka、Hadoop、Flink、Storm等。
- **CLI**：kubectl和相关工具。
- **cluster-lifecycle**：部署和升级kubernetes集群。
- **cluster-ops**：促进kubernetes集群本身的可操作性和集群间的互操作性，使不同的运营商之间协调一致。
- **contributor-experience**：维持良好的开发者社区。
- **docs**：文档，流程和出版物。
- **GCP**：在Google Cloud Platform上使用kubernetes。
- **instrumentation**：集群可观测性的最佳实践，包括指标设置、日志收集、事件等。
- **multicluster**：多kubernetes集群的用例和工具。
- **network**：kubernetes集群的网络。
- **node**：node节点、kubelet。
- **onprem**：在非云供应商的环境下运行kubernetes，例如on premise、裸机等环境。
- **openstack**：协调跨OpenStack和Kubernetes社区的努力。
- **product-management**：侧重于产品管理方面。
- **release**：发布、PR和bug提交等。
- **scalability**：负责回答可伸缩性相关的问题。
- **scheduling**：资源调度。
- **service-catalog**：为CNCF service broker和Kubernetes broker实现开发API。
- **storage**：存储和volume插件。
- **testing**：测试。
- **ui**：与UI相关的话题。

- **windows**：在kubernets上运行Windows Server Container。

工作组列表

- **App Def**：改进API中的声明性原语、客户端库、工具的用户体验。
- **Cloud Provider**：云供应商工作组。
- **Cluster API**：定义一个代表Kubernetes集群的可移植API。 API将包含控制平面及其配置和底层基础设施（节点，节点池等）。
- **Container Identity**：确保容器能够获得安全的身份认证并与外部连通的解决方案。
- **Kubeadm Adoption**：提高kubeadm工具的采用率。
- **Resource ManagementJ**：资源隔离和提高资源利用率。

详细信息请参考 https://github.com/kubernetes/community/blob/master/sig-list.md

配置Kubernetes开发环境

我们将在Mac上使用docker环境编译kuberentes。

安装依赖

```
brew install gnu-tar
```

Docker环境，至少需要给容器分配4G内存，在低于3G内存的时候可能会编译失败。

执行编译

切换目录到kuberentes源码的根目录下执行：

`./build/run.sh make` 可以在docker中执行跨平台编译出二进制文件。

需要用的的docker镜像：

```
gcr.io/google_containers/kube-cross:v1.7.5-2
```

我将该镜像备份到时速云上了，可供大家使用：

```
index.tenxcloud.com/jimmy/kube-cross:v1.7.5-2
```

该镜像基于Ubuntu构建，大小2.15G，编译环境中包含以下软件：

- Go1.7.5
- etcd
- protobuf
- g++
- 其他golang依赖包

在我自己的电脑上的整个编译过程大概要半个小时。

编译完成的二进制文件在 `/_output/local/go/bin/` 目录下。

本地分布式开发环境搭建（使用Vagrant和Virtualbox）

当我们需要在本地开发时，更希望能够有一个开箱即用又可以方便定制的分布式开发环境，这样才能对Kubernetes本身和应用进行更好的测试。现在我们使用Vagrant和VirtualBox来创建一个这样的环境。

部署时需要使用的配置文件和 `vagrantfile` 请见：https://github.com/rootsongjc/kubernetes-vagrant-centos-cluster

注意：kube-proxy使用ipvs模式。

准备环境

需要准备以下软件和环境：

- 8G以上内存
- Vagrant 2.0+
- Virtualbox 5.0 +
- 提前下载kubernetes的安装包

集群

我们使用Vagrant和Virtualbox安装包含3个节点的kubernetes集群，其中master节点同时作为node节点。

IP	主机名	组件
172.17.8.101	node1	kube-apiserver、kube-controller-manager、kube-scheduler、etcd、kubelet、docker、flannel
172.17.8.102	node2	kubelet、docker、flannel
172.17.8.103	node3	kubelet、docker、flannel

注意：以上的IP、主机名和组件都是固定在这些节点的，即使销毁后下次使用vagrant重建依然保持不变。

安装的组件

安装完成后的集群包含以下组件：

- flannel
- kubernetes dashboard
- etcd（单节点）
- kubectl

部署

确保安装好以上的准备环境后，执行下列命令启动kubernetes集群：

```
git clone https://github.com/rootsongjc/kubernetes-vagrant-cento
s-cluster.git
cd kubernetes-vagrant-centos-cluster
vagrant up
```

注意：克隆完Git仓库后，需要提前下载kubernetes的压缩包到 `kubenetes-vagrant-centos-cluster` 目录下，包括如下两个文件：

- kubernetes-client-linux-amd64.tar.gz
- kubernetes-server-linux-amd64.tar.gz

如果是首次部署，会自动下载 `centos/7` 的box，这需要花费一些时间，另外每个节点还需要下载安装一系列软件包，整个过程大概需要10几分钟。

访问kubernetes集群

访问Kubernetes集群的方式有三种：

通过本地访问

可以直接在你自己的本地环境中操作该kubernetes集群，而无需登录到虚拟机中，执行以下步骤：

将 `conf/admin.kubeconfig` 文件放到 `~/.kube/config` 目录下即可在本地使用 `kubectl` 命令操作集群。

在虚拟机内部访问

如果有任何问题可以登录到虚拟机内部调试：

```
vagrant ssh node1
sudo -i
kubectl get nodes
```

Kubernetes dashboard

还可以直接通过dashboard UI来访问：https://172.17.8.101:8443

可以在本地执行以下命令获取token的值（需要提前安装kubectl）：

```
kubectl -n kube-system describe secret `kubectl -n kube-system get secret|grep admin-token|cut -d " " -f1`|grep "token:"|tr -s " "|cut -d " " -f2
```

注意：token的值也可以在 `vagrant up` 的日志的最后看到。

Heapster监控

创建Heapster监控：

```
kubectl apply -f addon/heapster/
```

访问Grafana

使用Ingress方式暴露的服务，在本地 `/etc/hosts` 中增加一条配置：

```
172.17.8.102 grafana.jimmysong.io
```

访问Grafana：http://grafana.jimmysong.io

Traefik

部署Traefik ingress controller和增加ingress配置：

```
kubectl apply -f addon/traefik-ingress
```

在本地 `/etc/hosts` 中增加一条配置：

```
172.17.8.102 traefik.jimmysong.io
```

访问Traefik UI：http://traefik.jimmysong.io

EFK

使用EFK做日志收集。

```
kubectl apply -f addon/efk/
```

注意：运行EFK的每个节点需要消耗很大的CPU和内存，请保证每台虚拟机至少分配了4G内存。

清理

```
vagrant destroy
rm -rf .vagrant
```

参考

- Kubernetes handbook - jimmysong.io
- duffqiu/centos-vagrant
- kubernetes-vagrant-centos-cluster
- vagrant系列二：Vagrant的配置文件vagrantfile详解
- vagrant网络配置
- Kubernetes 1.8 kube-proxy 开启 ipvs

Kubernetes测试

单元测试

单元测试仅依赖于源代码，是测试代码逻辑是否符合预期的最简单方法。

运行所有的单元测试

```
make test
```

仅测试指定的**package**

```
# 单个package
make test WHAT=./pkg/api
# 多个packages
make test WHAT=./pkg/{api,kubelet}
```

或者，也可以直接用 `go test`

```
go test -v k8s.io/kubernetes/pkg/kubelet
```

仅测试指定**package**的某个测试**case**

```
# Runs TestValidatePod in pkg/api/validation with the verbose fl
ag set
make test WHAT=./pkg/api/validation KUBE_GOFLAGS="-v" KUBE_TEST_
ARGS='-run ^TestValidatePod$'

# Runs tests that match the regex ValidatePod|ValidateConfigMap
in pkg/api/validation
make test WHAT=./pkg/api/validation KUBE_GOFLAGS="-v" KUBE_TEST_
ARGS="-run ValidatePod\|ValidateConfigMap$"
```

或者直接用 `go test`

```
go test -v k8s.io/kubernetes/pkg/api/validation -run ^TestValida
tePod$
```

并行测试

并行测试是root out flakes的一种有效方法：

```
# Have 2 workers run all tests 5 times each (10 total iterations
).
make test PARALLEL=2 ITERATION=5
```

生成测试报告

```
make test KUBE_COVER=y
```

Benchmark测试

```
go test ./pkg/apiserver -benchmem -run=XXX -bench=BenchmarkWatch
```

集成测试

Kubernetes集成测试需要安装etcd（只要按照即可，不需要启动），比如

```
hack/install-etcd.sh  # Installs in ./third_party/etcd
echo export PATH="\$PATH:$(pwd)/third_party/etcd" >> ~/.profile
 # Add to PATH
```

集成测试会在需要的时候自动启动etcd和kubernetes服务，并运行test/integration里面的测试。

运行所有集成测试

```
make test-integration  # Run all integration tests.
```

指定集成测试用例

```
# Run integration test TestPodUpdateActiveDeadlineSeconds with t
he verbose flag set.
make test-integration KUBE_GOFLAGS="-v" KUBE_TEST_ARGS="-run ^Te
stPodUpdateActiveDeadlineSeconds$"
```

End to end (e2e)测试

End to end (e2e) 测试模拟用户行为操作Kubernetes，用来保证Kubernetes服务或集群的行为完全符合设计预期。

在开启e2e测试之前，需要先编译测试文件，并设置KUBERNETES_PROVIDER（默认为gce）：

```
make WHAT='test/e2e/e2e.test'
make ginkgo
export KUBERNETES_PROVIDER=local
```

启动**cluster**，测试，最后停止**cluster**

```
# build Kubernetes, up a cluster, run tests, and tear everything
 down
go run hack/e2e.go -- -v --build --up --test --down
```

仅测试指定的用例

```
go run hack/e2e.go -v -test --test_args='--ginkgo.focus=Kubectl\
sclient\s\[k8s\.io\]\sKubectl\srolling\-update\sshould\ssupport\
srolling\-update\sto\ssame\simage\s\[Conformance\]$'
```

略过测试用例

```
go run hack/e2e.go -- -v --test --test_args="--ginkgo.skip=Pods.
*env
```

并行测试

```
# Run tests in parallel, skip any that must be run serially
GINKGO_PARALLEL=y go run hack/e2e.go --v --test --test_args="--g
inkgo.skip=\[Serial\]"

# Run tests in parallel, skip any that must be run serially and
keep the test namespace if test failed
GINKGO_PARALLEL=y go run hack/e2e.go --v --test --test_args="--g
inkgo.skip=\[Serial\] --delete-namespace-on-failure=false"
```

清理测试

```
go run hack/e2e.go -- -v --down
```

有用的 `-ctl`

```
# -ctl can be used to quickly call kubectl against your e2e clus
ter. Useful for
# cleaning up after a failed test or viewing logs. Use -v to avo
id suppressing
# kubectl output.
go run hack/e2e.go -- -v -ctl='get events'
go run hack/e2e.go -- -v -ctl='delete pod foobar'
```

Fedaration e2e测试

```
export FEDERATION=true
export E2E_ZONES="us-central1-a us-central1-b us-central1-f"
# or export FEDERATION_PUSH_REPO_BASE="quay.io/colin_hom"
export FEDERATION_PUSH_REPO_BASE="gcr.io/${GCE_PROJECT_NAME}"

# build container images
KUBE_RELEASE_RUN_TESTS=n KUBE_FASTBUILD=true go run hack/e2e.go
-- -v -build

# push the federation container images
build/push-federation-images.sh

# Deploy federation control plane
go run hack/e2e.go -- -v --up

# Finally, run the tests
go run hack/e2e.go -- -v --test --test_args="--ginkgo.focus=\[Fe
ature:Federation\]"

# Don't forget to teardown everything down
go run hack/e2e.go -- -v --down
```

可以用 `cluster/log-dump.sh <directory>` 方便的下载相关日志，帮助排查测试中碰到的问题。

Node e2e测试

Node e2e仅测试Kubelet的相关功能，可以在本地或者集群中测试

```
export KUBERNETES_PROVIDER=local
make test-e2e-node FOCUS="InitContainer"
make test_e2e_node TEST_ARGS="--experimental-cgroups-per-qos=true"
```

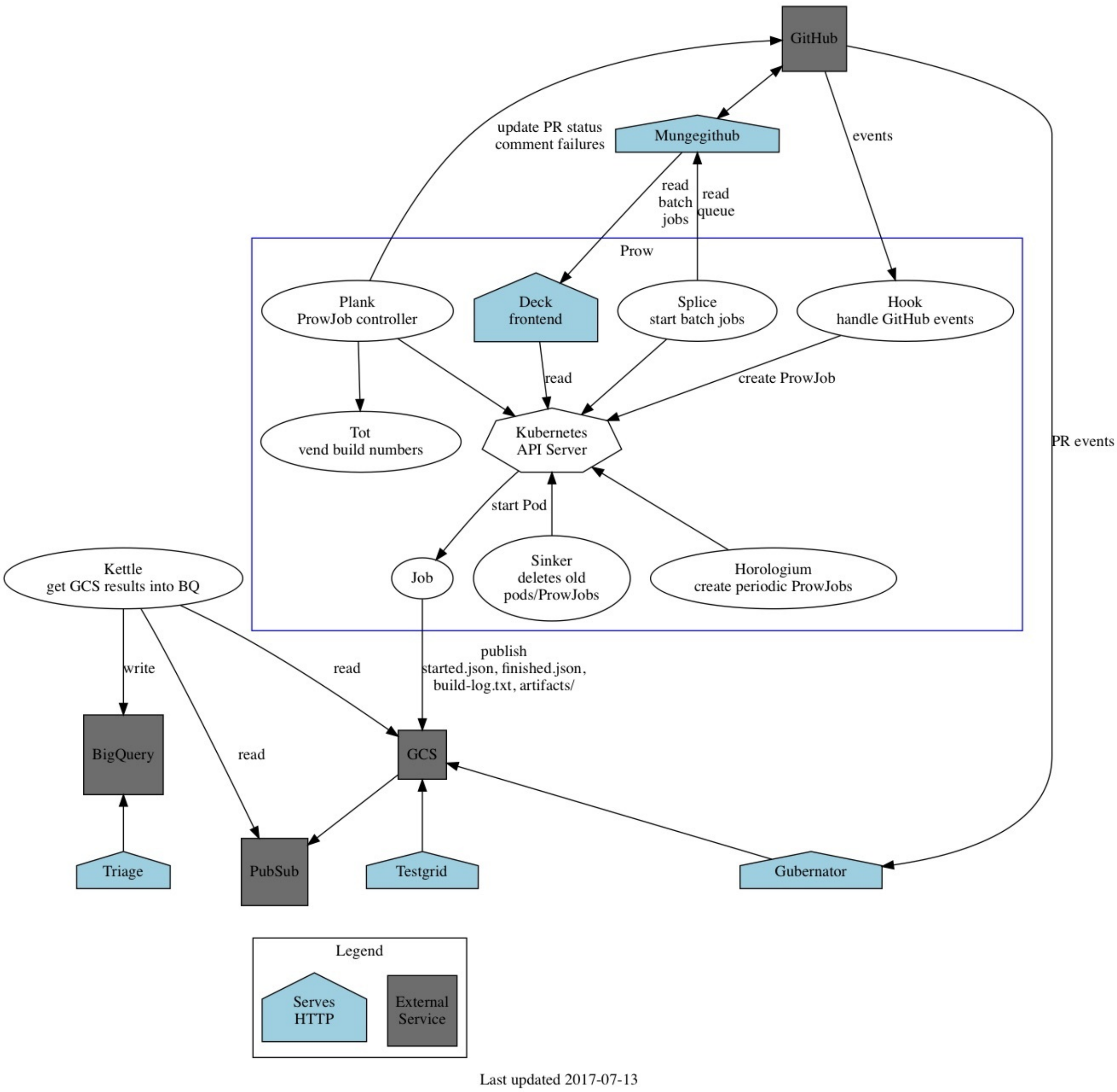
补充说明

借助kubectl的模版可以方便获取想要的数据，比如查询某个container的镜像的方法为

```
kubectl get pods nginx-4263166205-ggst4 -o template '--template={{if (exists . "status" "containerStatuses")}}{{range .status.containerStatuses}}{{if eq .name "nginx"}}{{.image}}{{end}}{{end}}{{end}}'
```

kubernetes测试工具集test-infra

test-infra是由kubernetes官方开源的测试框架，其中包括了Kubernetes测试工具集和测试结果展示。下图展示了test-infra的架构：



图片 - *test-infra*架构图（图片来自官方*GitHub*）

该测试框架主要是真多Google公有云做的，支持kubernetes1.6以上版本的测试。详见https://github.com/kubernetes/test-infra。

参考文档

- Kubernetes testing
- End-to-End Testing
- Node e2e test
- How to write e2e test

- Coding Conventions
- https://github.com/kubernetes/test-infra

client-go示例

访问kubernetes集群有几下几种方式：

方式	特点	支持者
Kubernetes dashboard	直接通过Web UI进行操作，简单直接，可定制化程度低	官方支持
kubectl	命令行操作，功能最全，但是比较复杂，适合对其进行进一步的分装，定制功能，版本适配最好	官方支持
client-go	从kubernetes的代码中抽离出来的客户端包，简单易用，但需要小心区分kubernetes的API版本	官方支持
client-python	python客户端，kubernetes-incubator	官方支持
Java client	fabric8中的一部分，kubernetes的java客户端	redhat

下面，我们基于client-go，对Deployment升级镜像的步骤进行了定制，通过命令行传递一个Deployment的名字、应用容器名和新image名字的方式来升级。代码和使用方式见 https://github.com/rootsongjc/kubernetes-client-go-sample 。

kubernetes-client-go-sample

代码如下：

```
package main

import (
	"flag"
	"fmt"
	"os"
	"path/filepath"

	"k8s.io/apimachinery/pkg/api/errors"
	metav1 "k8s.io/apimachinery/pkg/apis/meta/v1"
	"k8s.io/client-go/kubernetes"
	"k8s.io/client-go/tools/clientcmd"
)

func main() {
	var kubeconfig *string
```

```
    if home := homeDir(); home != "" {
        kubeconfig = flag.String("kubeconfig", filepath.Join(hom
e, ".kube", "config"), "(optional) absolute path to the kubeconf
ig file")
    } else {
        kubeconfig = flag.String("kubeconfig", "", "absolute pat
h to the kubeconfig file")
    }
    deploymentName := flag.String("deployment", "", "deployment
name")
    imageName := flag.String("image", "", "new image name")
    appName := flag.String("app", "app", "application name")

    flag.Parse()
    if *deploymentName == "" {
        fmt.Println("You must specify the deployment name.")
        os.Exit(0)
    }
    if *imageName == "" {
        fmt.Println("You must specify the new image name.")
        os.Exit(0)
    }
    // use the current context in kubeconfig
    config, err := clientcmd.BuildConfigFromFlags("", *kubeconfi
g)
    if err != nil {
        panic(err.Error())
    }

    // create the clientset
    clientset, err := kubernetes.NewForConfig(config)
    if err != nil {
        panic(err.Error())
    }
    deployment, err := clientset.AppsV1beta1().Deployments("defa
ult").Get(*deploymentName, metav1.GetOptions{})
    if err != nil {
        panic(err.Error())
    }
    if errors.IsNotFound(err) {
        fmt.Printf("Deployment not found\n")
    } else if statusError, isStatus := err.(*errors.StatusError)
; isStatus {
        fmt.Printf("Error getting deployment%v\n", statusError.E
rrStatus.Message)
    } else if err != nil {
        panic(err.Error())
    } else {
        fmt.Printf("Found deployment\n")
        name := deployment.GetName()
        fmt.Println("name ->", name)
        containers := &deployment.Spec.Template.Spec.Containers
        found := false
```

```
        for i := range *containers {
            c := *containers
            if c[i].Name == *appName {
                found = true
                fmt.Println("Old image ->", c[i].Image)
                fmt.Println("New image ->", *imageName)
                c[i].Image = *imageName
            }
        }
        if found == false {
            fmt.Println("The application container not exist in
the deployment pods.")
            os.Exit(0)
        }
        _, err := clientset.AppsV1beta1().Deployments("default")
.Update(deployment)
        if err != nil {
            panic(err.Error())
        }
    }
}

func homeDir() string {
    if h := os.Getenv("HOME"); h != "" {
        return h
    }
    return os.Getenv("USERPROFILE") // windows
}
```

我们使用 `kubeconfig` 文件认证连接kubernetes集群，该文件默认的位置是 `$HOME/.kube/config` 。

该代码编译后可以直接在kubernetes集群之外，任何一个可以连接到API server的机器上运行。

编译运行

```
$ go get github.com/rootsongjc/kubernetes-client-go-sample
$ cd $GOPATH/src/github.com/rootsongjc/kubernetes-client-go-samp
le
$ make
$ ./update-deployment-image -h
Usage of ./update-deployment-image:
  -alsologtostderr
        log to standard error as well as files
  -app string
        application name (default "app")
  -deployment string
        deployment name
  -image string
        new image name
  -kubeconfig string
        (optional) absolute path to the kubeconfig file (default
"/Users/jimmy/.kube/config")
  -log_backtrace_at value
        when logging hits line file:N, emit a stack trace
  -log_dir string
        If non-empty, write log files in this directory
  -logtostderr
        log to standard error instead of files
  -stderrthreshold value
        logs at or above this threshold go to stderr
  -v value
        log level for V logs
  -vmodule value
        comma-separated list of pattern=N settings for file-filt
ered logging
```

使用不存在的image更新

```
 $ ./update-deployment-image -deployment filebeat-test -image ha
rbor-001.jimmysong.io/library/analytics-docker-test:Build_9
Found deployment
name -> filebeat-test
Old image -> harbor-001.jimmysong.io/library/analytics-docker-te
st:Build_8
New image -> harbor-001.jimmysong.io/library/analytics-docker-te
st:Build_9
```

查看Deployment的event。

```
$ kubectl describe deployment filebeat-test
Name:            filebeat-test
Namespace:        default
CreationTimestamp:    Fri, 19 May 2017 15:12:28 +0800
Labels:             k8s-app=filebeat-test
Selector:          k8s-app=filebeat-test
Replicas:          2 updated | 3 total | 2 available | 2 unavailab
le
StrategyType:         RollingUpdate
MinReadySeconds:    0
RollingUpdateStrategy:    1 max unavailable, 1 max surge
Conditions:
  Type        Status    Reason
  ----        ------    ------
  Available     True    MinimumReplicasAvailable
  Progressing     True    ReplicaSetUpdated
OldReplicaSets:    filebeat-test-2365467882 (2/2 replicas create
d)
NewReplicaSet:    filebeat-test-2470325483 (2/2 replicas created
)
Events:
  FirstSeen    LastSeen    Count    From                SubObjec
tPath    Type        ReasoMessage
  ---------    --------    -----    ----                --------
-----    --------    ------------
  2h        1m        3    {deployment-controller }            N
ormal        ScalingReplicaSet    Scaled down replica set filebe
at-test-2365467882 to 2
  1m        1m        1    {deployment-controller }            N
ormal        ScalingReplicaSet    Scaled up replica set filebeat
-test-2470325483 to 1
  1m        1m        1    {deployment-controller }            N
ormal        ScalingReplicaSet    Scaled up replica set filebeat
-test-2470325483 to 2
```

可以看到老的ReplicaSet从3个replica减少到了2个，有2个使用新配置的replica不可用，目前可用的replica是2个。

这是因为我们指定的镜像不存在，查看Deployment的pod的状态。

```
$ kubectl get pods -l k8s-app=filebeat-test
NAME                                 READY     STATUS             RE
STARTS   AGE
filebeat-test-2365467882-4zwx8   2/2       Running            0
         33d
filebeat-test-2365467882-rqskl   2/2       Running            0
         33d
filebeat-test-2470325483-6vjbw   1/2       ImagePullBackOff   0
         4m
filebeat-test-2470325483-gc14k   1/2       ImagePullBackOff   0
         4m
```

我们可以看到有两个pod正在拉取image。

还原为原先的镜像

将image设置为原来的镜像。

```
$ ./update-deployment-image -deployment filebeat-test -image har
bor-001.jimmysong.io/library/analytics-docker-test:Build_8
Found deployment
name -> filebeat-test
Old image -> harbor-001.jimmysong.io/library/analytics-docker-te
st:Build_9
New image -> harbor-001.jimmysong.io/library/analytics-docker-te
st:Build_8
```

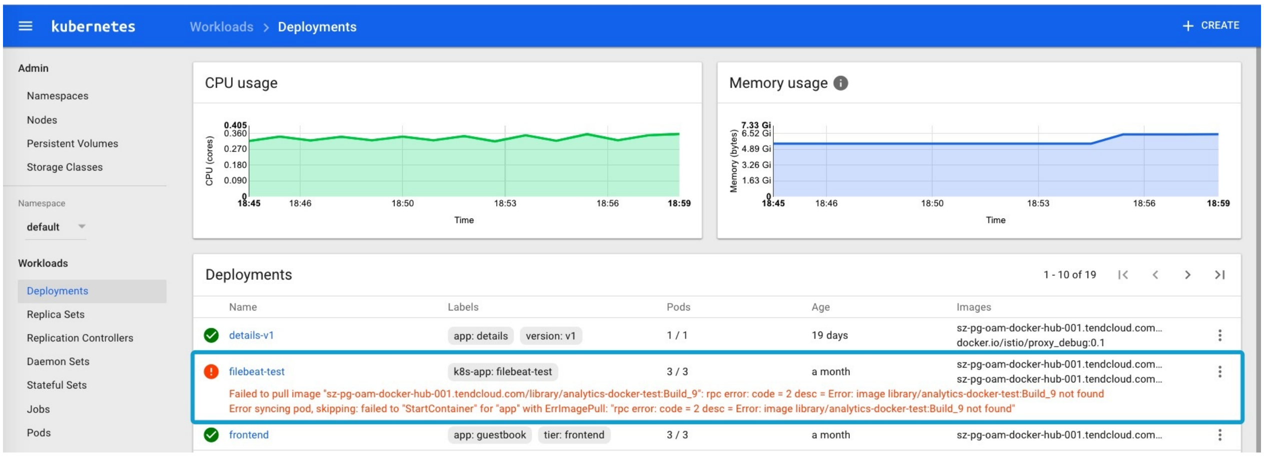
现在再查看Deployment的状态。

```
$ kubectl describe deployment filebeat-test
Name:                   filebeat-test
Namespace:              default
CreationTimestamp:      Fri, 19 May 2017 15:12:28 +0800
Labels:                 k8s-app=filebeat-test
Selector:               k8s-app=filebeat-test
Replicas:               3 updated | 3 total | 3 available | 0 unavailab
le
StrategyType:           RollingUpdate
MinReadySeconds:        0
RollingUpdateStrategy:  1 max unavailable, 1 max surge
Conditions:
  Type          Status  Reason
  ----          ------  ------
  Available     True    MinimumReplicasAvailable
  Progressing   True    NewReplicaSetAvailable
OldReplicaSets:         <none>
NewReplicaSet:          filebeat-test-2365467882 (3/3 replicas created)

Events:
  FirstSeen     LastSeen    Count   From                    SubObjec
tPath    Type        ReasoMessage
  ---------     --------    -----   ----                    --------
-----    --------    ------------
  2h        8m      3   {deployment-controller }            N
ormal       ScalingReplicaSet   Scaled down replica set filebe
at-test-2365467882 to 2
  8m        8m      1   {deployment-controller }            N
ormal       ScalingReplicaSet   Scaled up replica set filebeat
-test-2470325483 to 1
  8m        8m      1   {deployment-controller }            N
ormal       ScalingReplicaSet   Scaled up replica set filebeat
-test-2470325483 to 2
  2h        1m      3   {deployment-controller }            N
ormal       ScalingReplicaSet   Scaled up replica set filebeat
-test-2365467882 to 3
  1m        1m      1   {deployment-controller }            N
ormal       ScalingReplicaSet   Scaled down replica set filebe
at-test-2470325483 to 0
```

可以看到available的replica个数恢复成3了。

其实在使用该命令的过程中，通过kubernetes dashboard的页面上查看Deployment的状态更直观，更加方便故障排查。



图片 - 使用*kubernetes dashboard*进行故障排查

这也是dashboard最大的优势，简单、直接、高效。

Operator

Operator是由CoreOS开发的，用来扩展Kubernetes API，特定的应用程序控制器，它用来创建、配置和管理复杂的有状态应用，如数据库、缓存和监控系统。Operator基于Kubernetes的资源和控制器概念之上构建，但同时又包含了应用程序特定的领域知识。创建Operator的关键是CRD（自定义资源）的设计。

工作原理

Operator是将运维人员对软件操作的知识给代码化，同时利用Kubernetes强大的抽象来管理大规模的软件应用。

Operator使用了Kubernetes的自定义资源扩展API机制，如使用CRD（CustomResourceDefinition）来创建。Operator通过这种机制来创建、配置和管理应用程序。

当前CoreOS提供的以下四种Operator：

- etcd：创建etcd集群
- Rook：云原生环境下的文件、块、对象存储服务
- Prometheus：创建Prometheus监控实例
- Tectonic：部署Kubernetes集群

以上为CoreOS官方提供的Operator，另外还有很多很多其他开源的Operator实现。

Operator基于Kubernetes的以下两个概念构建：

- 资源：对象的状态定义
- 控制器：观测、分析和行动，以调节资源的分布

创建Operator

Operator本质上是与应用息息相关的，因为这是特定领域的知识的编码结果，这其中包括了资源配置的控制逻辑。下面是创建Operator的基本套路：

1. 在单个Deployment中定义Operator，

如：https://coreos.com/operators/etcd/latest/deployment.yaml
2. 需要为Operator创建一个新的自定义类型CRD，这样用户就可以使用该对象来创建实例
3. Operator应该利用Kubernetes中内建的原语，如Deployment、Service这些经过充分测试的对象，这样也便于理解
4. Operator应该向后兼容，始终了解用户在之前版本中创建的资源
5. 当Operator被停止或删除时，Operator创建的应用实例应该不受影响
6. Operator应该让用户能够根据版本声明来选择所需版本和编排应用程序升级。不升级软件是操作错误和安全问题的常见来源，Operator可以帮助用户更加自信地解决这一问题。
7. Operator应该进行"Chaos Monkey"测试，以模拟Pod、配置和网络故障的情况下的行为。

参考

- Operators - coreos.com
- Writing a Kubernetes Operator in Golang
- Introducing Operators: Putting Operational Knowledge into Software
- Automating Kubernetes Cluster Operations with Operators

Operator SDK

Operator SDK 由 CoreOS 开源，它是用于构建 Kubernetes 原生应用的 SDK，它提供更高级别的 API、抽象和项目脚手架。在阅读本文前请先确认您已经了解 Operator是什么。

使用 Kubernetes 中原生的对象来部署和管理复杂的应用程序不是那么容易，尤其是要管理整个应用的生命周期、组件的扩缩容，我们之前通常是编写各种脚本，通过调用 Kubernetes 的命令行工具来管理 Kubernetes 上的应用。现在可以通过 CRD（CustomResourceDefinition）来自定义这些复杂操作，通过将运维的知识封装在自定义 API 里来减轻运维人员的负担。同时我们还可以像操作 Kubernetes 的原生资源对象一样，使用 `kubectl` 来操作 CRD。

下面我们将安装和试用一下 Operator SDK。

安装 Operator SDK

```
$ mkdir -p $GOPATH/src/github.com/operator-framework
$ cd $GOPATH/src/github.com/operator-framework/operator-sdk
$ dep ensure
Create kubernetes-operator-sdk-tutorial/cmd/kubernetes-operator-sdk-tutorial/main.go
Create kubernetes-operator-sdk-tutorial/config/config.yaml
Create kubernetes-operator-sdk-tutorial/deploy/rbac.yaml
Create kubernetes-operator-sdk-tutorial/deploy/cr.yaml
Create kubernetes-operator-sdk-tutorial/pkg/apis/jimmysong/v1alpha1/doc.go
Create kubernetes-operator-sdk-tutorial/pkg/apis/jimmysong/v1alpha1/register.go
Create kubernetes-operator-sdk-tutorial/pkg/apis/jimmysong/v1alpha1/types.go
Create kubernetes-operator-sdk-tutorial/pkg/stub/handler.go
Create kubernetes-operator-sdk-tutorial/tmp/build/build.sh
Create kubernetes-operator-sdk-tutorial/tmp/build/docker_build.sh
Create kubernetes-operator-sdk-tutorial/tmp/build/Dockerfile
Create kubernetes-operator-sdk-tutorial/tmp/codegen/boilerplate.go.txt
Create kubernetes-operator-sdk-tutorial/tmp/codegen/update-generated.sh
Create kubernetes-operator-sdk-tutorial/Gopkg.toml
Create kubernetes-operator-sdk-tutorial/Gopkg.lock
Run dep ensure ...
Root project is "github.com/rootsongjc/kubernetes-operator-sdk-tutorial"
 3 transitively valid internal packages
 12 external packages imported from 4 projects
(0)   ✓ select (root)
(1)    ? attempt k8s.io/api with 1 pkgs; at least 1 versions to try
(1)        try k8s.io/api@kubernetes-1.9.3
(1)    ✓ select k8s.io/api@kubernetes-1.9.3 w/1 pkgs
(2)    ? attempt k8s.io/apimachinery with 4 pkgs; at least 1 versions to try
(2)        try k8s.io/apimachinery@kubernetes-1.9.3
(2)    ✓ select k8s.io/apimachinery@kubernetes-1.9.3 w/22 pkgs
...

$ go install github.com/operator-framework/operator-sdk/commands/operator-sdk
```

该过程需要几分钟，请耐心等待。确认 `$GOPATH/bin/operator-sdk` 文件位于您的 `$PATH` 目录下。

创建项目

```
$ cd $GOPATH/src/github.com/<your-github-repo>/
$ operator-sdk new <operator-project-name> --api-version=<your-api-group>/<version> --kind=<custom-resource-kind>
$ cd <operator-project-name>
```

- operator-project-name：创建的项目的名称
- your-api-group：Kubernetes 自定义 API 的组名，一般用域名如 `jimmysong.io`
- version：Kubernetes 自定义资源的 API 版本
- custom-resource-kind：CRD 的名称

```
operator-sdk new kubernetes-operator-sdk-tutorial --api-version=jimmysong.io/v1alpha1 --kind=operator-sdk
```

参考

- A complete guide to Kubernetes Operator SDK

高级开发指南

本页假定您已经熟悉 Kubernetes 的核心概念并可以轻松的部署自己的应用程序。如果还不能，您需要先查看下中级应用开发者主题。

在浏览了本页面及其链接的内容后，您将会更好的理解如下部分：

- 可以在应用程序中使用的高级功能
- 扩展 Kubernetes API 的各种方法

使用高级功能部署应用

现在您知道了 Kubernetes 中提供的一组 API 对象。理解了 daemonset 和 deployment 之间的区别对于应用程序部署通常是足够的。也就是说，熟悉 Kubernetes 中其它的鲜为人知的功能也是值得的。因为这些功能有时候对于特别的用例是非常强大的。

容器级功能

如您所知，将整个应用程序（例如容器化的 Rails 应用程序，MySQL 数据库以及所有应用程序）迁移到单个 Pod 中是一种反模式。这就是说，有一些非常有用的模式超出了容器和 Pod 之间的 1:1 的对应关系：

- **Sidecar** 容器：虽然 Pod 中依然需要有一个主容器，你还可以添加一个副容器作为辅助（见 日志示例)。单个 *Pod* 中的两个容器可以通过共享卷进行通信。
- **Init** 容器：*Init* 容器在 Pod 的应用容器（如主容器和 sidecar 容器）之前运行。阅读更多，查看 nginx 服务器示例，并学习如何调试这些容器。

Pod 配置

通常，您可以使用 label 和 annotation 将元数据附加到资源上。将数据注入到资源，您可以会创建 ConfigMap（用于非机密数据）或 Secret（用于机密数据）。

下面是一些其他不太为人所知的配置资源 Pod 的方法：

- Taint（污点）和 Toleration（容忍）：这些为节点“吸引”或“排斥” Pod 提供了一种方法。当需要将应用程序部署到特定硬件（例如用于科学计算的 GPU）

时，经常使用它们。阅读更多。

- **向下 API**：这允许您的容器使用有关自己或集群的信息，而不会过度耦合到 Kubernetes API server。这可以通过环境变量 或者 DownwardAPIVolumeFiles。
- **Pod 预设**：通常，要将运行时需求（例如环境变量、ConfigMap 和 Secret）安装到资源中，可以在资源的配置文件中指定它们。*PodPresets*允许您在创建资源时动态注入这些需求。例如，这允许团队 *A* 将任意数量的新*Secret* 安装到团队 *B* 和 *C* 创建的资源中，而不需要 *B* 和 *C* 的操作。请参阅示例。

其他 API 对象

在设置以下资源之前，请检查这是否属于您组织的集群管理员的责任。

- **Horizontal Pod Autoscaler (HPA)** ：这些资源是在CPU使用率或其他自定义度量标准“秒杀”时自动化扩展应用程序的好方法。查看示例以了解如何设置 HPA。
- 联合集群对象：如果使用 *federation* 在多个 Kubernetes 集群上运行应用程序，则需要部署标准 Kubernetes API 对象的联合版本。有关参考，请查看设置 联合 ConfigMap 和联合 Deployment 的指南。

扩展 Kubernetes API

Kubernetes 在设计之初就考虑到了可扩展性。如果上面提到的 API 资源和功能不足以满足您的需求，则可以自定义其行为，而无需修改核心 Kubernetes 代码。

理解 Kubernetes 的默认行为

在进行任何自定义之前，了解 Kubernetes API 对象背后的一般抽象很重要。虽然 Deployment 和 Secret 看起来可能完全不同，但对于任何对象来说，以下概念都是正确的：

- **Kubernetes**对象是存储有关您的集群的结构化数据的一种方式。

  在 Deployment 的情况下，该数据代表期望的状态（例如“应该运行多少副本？”），但也可以是通用的元数据（例如数据库凭证）。

- **Kubernetes** 对象通过 **Kubernetes API** 修改。

换句话说，您可以对特定的资源路径（例如 `<api-server-url>/api/v1/namespaces/default/deployments` ）执行 `GET` 和 `POST` 请求来读取或修改对应的对象类型。

- 利用 **Controller 模式**，**Kubernetes** 对象可被确保达到期望的状态。为了简单起见，您可以将 Controller 模式看作以下连续循环：

  1. 检查当前状态（副本数、容器镜像等）
  2. 对比当前状态和期望状态
  3. 如果不匹配则更新当前状态

  这些状态是通过 Kubernetes API 来获取的。

  并非所有的 Kubernetes 对象都需要一个 Controller。尽管 Deployment 触发群集进行状态更改，但 ConfigMaps 纯粹作为存储。

创建自定义资源

基于上述想法，您可以定义与 Deployment 一样合法的自定义资源。例如，如果 `CronJobs` 不能提供所有您需要的功能，您可能需要定义 `Backup` 对象以进行定期备份。

创建自定义资源有以下两种方式：

1. 自定义资源定义（**CRD**）：这种实现方式的工作量最小。参考示例。
2. **API** 聚合：在实际设置单独的扩展 API server 之前，此方法需要一些预配置

请注意，与依赖内置的 `kube-controller-manager` 不同，您需要编写并运行自定义控制器。

下面是一些有用的链接：

- 如何才知道自定义资源是否符合您的使用场景
- CRD 还是 API 聚合，如何选择？

Service Catalog

如果您想要使用或提供完整的服务（而不是单个资源），**Service Catalog** 为此提供了一个规范。这些服务使用 Service Broker注册（请参阅 示例）。

如果您没有集群管理员来管理 Service Catalog 的安装，您可以使用 Helm 或 二进制安装器。

探索其他资源

参考

以下主题对构建更复杂的应用程序也很有用：

- Kubernetes 中的其他扩展点 - 在哪里可以挂勾到 Kubernetes 架构的概念性的概述
- Kubernetes 客户端库 - 用于构建需要与 Kubernetes API 大量交互的应用程序。

下一步

恭喜您完成了应用开发者之旅！您已经了解了 Kubernetes 提供的大部分功能。现在怎么办？

- 如果您想推荐新功能或跟上Kubernetes应用开发的最新进展，请考虑加入 SIG，如 SIG Apps。
- 如果您有兴趣详细了解 Kubernetes 的内部运作（例如网络），请考虑查看集群运维之旅。

原文：https://kubernetes.iohttps://kubernetes.io/docs/user-journeys/users/application-developer/advanced

Kubernetes社区贡献

- Contributing guidelines
- Kubernetes Developer Guide
- Special Interest Groups
- Feature Tracking and Backlog
- Community Expectations
- Kubernetes 官方文档汉化项目

Minikube

Minikube用于在本地运行kubernetes环境，用来开发和测试。

在Mac上安装xhyve-driver

```
brew install docker-machine-driver-xhyve
# docker-machine-driver-xhyve need root owner and uid
sudo chown root:wheel $(brew --prefix)/opt/docker-machine-driver-xhyve/bin/docker-machine-driver-xhyve
sudo chmod u+s $(brew --prefix)/opt/docker-machine-driver-xhyve/bin/docker-machine-driver-xhyve
```

到 https://github.com/kubernetes/minikube/releases 下载 minikube，我安装的是 minikube v0.22.3

下载完成后修改文件名为 `minikube` ，然后 `chmod +x minikube` ，移动到 `$PATH` 目录下：

```
mv ~/Download/minikube-darwin-adm64 /usr/loal/bin/
chmod +x /usr/local/bin/minikube
```

安装kubectl

参考Install and Set Up kubectl，直接使用二进制文件安装即可。

```
curl -LO https://storage.googleapis.com/kubernetes-release/release/`curl -s https://storage.googleapis.com/kubernetes-release/release/stable.txt`/bin/darwin/amd64/kubectl
或者：
先访问https://storage.googleapis.com/kubernetes-release/release/stable.txt
得到返回值，假设为:v1.9.1，然后拼接网址，直接在浏览器访问：
https://storage.googleapis.com/kubernetes-release/release/v1.9.1/bin/darwin/amd64/kubectl
直接下载kubectl文件。
```

若第一种方式访问多次超时，可以使用上述的第二种方式访问。

启动Minikube

假设使用xhyve-driver虚拟技术，则需要在minikube start加入参数 `--vm-driver=xhyve` 。

```
minikube start --vm-driver=xhyve
Starting local Kubernetes v1.7.5 cluster...
Starting VM...
Downloading Minikube ISO
 139.09 MB / 139.09 MB [========================================
====] 100.00% 0s
Getting VM IP address...
Moving files into cluster...
Setting up certs...
Connecting to cluster...
Setting up kubeconfig...
Starting cluster components...
Kubectl is now configured to use the cluster.
```

这将生成默认的 `~/.kube/config` 文件，自动指向minikube。

停止Minikube

```
minikube stop
```

参考

Running Kubernetes Locally via Minikube

Install minikube

Driver plugin installation - xhyve-driver

附录说明

参考文档以及一些实用的资源链接。

- Kubernetes documentation
- Awesome Kubernetes
- Kubernetes the hard way
- Kubernetes Bootcamp
- Design patterns for container-based distributed systems
- Awesome Cloud Native

Kubernetes service中的故障排查

- 查看某个资源的定义和用法

```
kubectl explain
```

- 查看Pod的状态

```
kubectl get pods
kubectl describe pods my-pod
```

- 监控Pod状态的变化

```
kubectl get pod -w
```

可以看到一个 namespace 中所有的 pod 的 phase 变化，请参考 Pod 的生命周期。

- 查看 Pod 的日志

```
kubectl logs my-pod
kubectl logs my-pod -c my-container
kubectl logs -f my-pod
kubectl logs -f my-pod -c my-container
```

`-f` 参数可以 follow 日志输出。

- 交互式 debug

```
kubectl exec my-pod -it /bin/bash
kubectl top pod POD_NAME --containers
```

Kubernetes相关资讯和情报链接

授人以鱼，不如授人以渔。下面的资料将有助于大家了解kubernetes生态圈当前发展状况和发展趋势，我特此整理相关资料如下。

生态环境

包括kubernetes和cloud native相关的开源软件、工具和全景图。

- awesome-kubernetes - A curated list for awesome kubernetes sources https://ramitsurana.github.io/awesome…
- awesome-cloud-native - A curated list for awesome cloud native architectures https://jimmysong.io/awesome-cloud-native/
- cloud native landscape - Cloud Native Landscape https://cncf.io

开源书籍和教程

- arun-gupta/kubernetes-aws-workshop
- arun-gupta/kubernetes-java-sample
- feiskyer/kubernetes-handbook
- kelseyhightower/kubernetes-the-hard-way
- ks - A series of Kubernetes walk-throughs
- opsnull/follow-me-install-kubernetes-cluster
- rootsongjc/kubernetes-handbook

幻灯片、图书和情报资料分享

- cloud-native-slides-share - Cloud Native 相关meetup、会议PPT、图书资料分享

博客与网站

Kubernetes和Cloud Native相关网站、专栏、博客等。

网站与专栏

- thenewstack.io
- k8sport.org
- giantswarm blog
- k8smeetup.com
- dockone.io
- Cloud Native知乎专栏
- kubernetes.org.cn

博客

- apcera
- aporeto
- applatix
- apprenda
- bitnami
- buoyant
- cisco
- cncf
- codeship
- containership
- coreos
- coscale
- deis
- fabric8
- grafana
- gravitational
- heptio
- istio
- jimmysong
- kubernetes
- moby
- openshift
- pivotal
- platform9

- prometheus
- rancher
- sysdig
- spinnaker
- supergiant
- thecodeteam
- twistlock
- vamp
- weave
- wercker

Docker最佳实践

本文档旨在实验Docker1.13新特性和帮助大家了解docker集群的管理和使用。

环境配置

Docker1.13环境配置

docker源码编译

网络管理

网络配置和管理是容器使用中的的一个重点和难点，对比我们之前使用的docker版本是1.11.1，docker1.13中网络模式跟之前的变动比较大，我们会花大力气讲解。

如何创建docker network

Rancher网络探讨和扁平网络实现

swarm mode的路由网络

docker扁平化网络插件Shrike（基于docker1.11）

存储管理

Docker存储插件

- infinit 被docker公司收购的法国团队开发
- convoy rancher开发的docker volume plugin
- torus 已废弃
- flocker ClusterHQ开发

日志管理

Docker提供了一系列log drivers，如fluentd、journald、syslog等。

需要配置docker engine的启动参数。

创建应用

官方文档：Docker swarm sample app overview

基于docker1.13手把手教你创建swarm app

swarm集群应用管理

使用docker-compose创建应用

集群管理

我们使用docker内置的swarm来管理docker集群。

swarm mode介绍

我们推荐使用开源的docker集群管理配置方案：

- Crane：由数人云开源的基于swarmkit的容器管理软件，可以作为docker和go语言开发的一个不错入门项目
- Rancher:Rancher是一个企业级的容器管理平台，可以使用Kubernetes、swarm和rancher自研的cattle来管理集群。

Crane的部署和使用

Rancher的部署和使用

资源限制

内存资源限制

CPU资源限制

IO资源限制

服务发现

下面罗列一些列常见的服务发现工具。

Etcd:服务发现/全局分布式键值对存储。这是CoreOS的创建者提供的工具，面向容器和宿主机提供服务发现和全局配置存储功能。它在每个宿主机有基于http协议的API和命令行的客户端。https://github.com/docker/etcd

- Cousul：服务发现/全局分布式键值对存储。这个服务发现平台有很多高级的特性，使得它能够脱颖而出，例如：配置健康检查、ACL功能、HAProxy配置等等。
- Zookeeper：诞生于Hadoop生态系统里的组件，Apache的开源项目。服务发现/全局分布式键值对存储。这个工具较上面两个都比较老，提供一个更加成熟的平台和一些新特性。
- Crypt：加密etcd条目的项目。Crypt允许组建通过采用公钥加密的方式来保护它们的信息。需要读取数据的组件或被分配密钥，而其他组件则不能读取数据。
- Confd：观测键值对存储变更和新值的触发器重新配置服务。Confd项目旨在基于服务发现的变化，而动态重新配置任意应用程序。该系统包含了一个工具来检测节点中的变化、一个模版系统能够重新加载受影响的应用。
- Vulcand：vulcand在组件中作为负载均衡使用。它使用etcd作为后端，并基于检测变更来调整它的配置。
- Marathon：虽然marathon主要是调度器，它也实现了一个基本的重新加载HAProxy的功能，当发现变更时，它来协调可用的服务。
- Frontrunner：这个项目嵌入在marathon中对HAproxy的更新提供一个稳定的解决方案。
- Synapse：由Airbnb出品的，Ruby语言开发，这个项目引入嵌入式的HAProxy组件，它能够发送流量给各个组件。http://bruth.github.io/synapse/docs/
- Nerve：它被用来与synapse结合在一起来为各个组件提供健康检查，如果组件不可用，nerve将更新synapse并将该组件移除出去。

插件开发

插件开发示例-sshfs

我的docker插件开发文章

Docker17.03-CE插件开发举例

网络插件

- Contiv 思科出的Docker网络插件，趟坑全记录，目前还无法上生产，1.0正式版还没出，密切关注中。
- Calico 产品化做的不错，已经有人用在生产上了。

存储插件

业界使用案例

京东从OpenStack切换到Kubernetes的经验之谈

美团点评容器平台介绍

阿里超大规模docker化之路

TalkingData-容器技术在大数据场景下的应用Yarn on Docker

乐视云基于Kubernetes的PaaS平台建设

资源编排

建议使用kuberentes，虽然比较复杂，但是专业的工具做专业的事情，将编排这么重要的生产特性绑定到docker上的风险还是很大的，我已经转投到kubernetes怀抱了，那么你呢？

我的kubernetes探险之旅

相关资源

容器技术工具与资源

容器技术2016年总结

关于

Author: Jimmy Song

rootsongjc@gmail.com

更多关于**Docker**、**MicroServices**、**Big Data**、**DevOps**、**Deep Learning**的内容请关注Jimmy Song's Blog，将不定期更新。

1. 在容器中获取 Pod 的IP

通过环境变量来实现，该环境变量直接引用 resource 的状态字段，示例如下：

```yaml
apiVersion: v1
kind: ReplicationController
metadata:
  name: world-v2
spec:
  replicas: 3
  selector:
    app: world-v2
  template:
    metadata:
      labels:
        app: world-v2
    spec:
      containers:
      - name: service
        image: test
        env:
        - name: POD_IP
          valueFrom:
            fieldRef:
              fieldPath: status.podIP
        ports:
        - name: service
          containerPort: 777
```

容器中可以直接使用 `POD_IP` 环境变量获取容器的 IP。

2. 指定容器的启动参数

我们可以在 Pod 中为容器使用 command 为容器指定启动参数：

```
command: ["/bin/bash","-c","bootstrap.sh"]
```

看似很简单，使用数组的方式定义，所有命令使用跟 Dockerfile 中的 CMD 配置是一样的，但是有一点不同的是， `bootsttap.sh` 必须具有可执行权限，否则容器启动时会出错。

3. 让Pod调用宿主机的docker能力

我们可以想象一下这样的场景，让 Pod 来调用宿主机的 docker 能力，只需要将宿主机的 `docker` 命令和 `docker.sock` 文件挂载到 Pod 里面即可，如下：

```
apiVersion: v1
kind: Pod
metadata:
 name: busybox-cloudbomb
spec:
 containers:
 - image: busybox
 command:
 - /bin/sh
 - "-c"
 - "while true; \
 do \
 docker run -d --name BOOM_$(cat /dev/urandom | tr -cd 'a-f0-9' |
 head -c 6) nginx ; \
 done"
 name: cloudbomb
 volumeMounts:
 - mountPath: /var/run/docker.sock
 name: docker-socket
 - mountPath: /bin/docker
 name: docker-binary
 volumes:
 - name: docker-socket
 hostPath:
 path: /var/run/docker.sock
 - name: docker-binary
 hostPath:
 path: /bin/docker
```

参考：Architecture Patterns for Microservices in Kubernetes

4. 使用Init container初始化应用配置

Init container可以在应用程序的容器启动前先按顺序执行一批初始化容器，只有所有Init容器都启动成功后，Pod才算启动成功。看下下面这个例子（来源：kubernetes: mounting volume from within init container - Stack Overflow）：

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: init
  labels:
    app: init
  annotations:
    pod.beta.kubernetes.io/init-containers: '[
        {
            "name": "download",
            "image": "axeclbr/git",
            "command": [
                "git",
                "clone",
                "https://github.com/mdn/beginner-html-site-scrip
ted",
                "/var/lib/data"
            ],
            "volumeMounts": [
                {
                    "mountPath": "/var/lib/data",
                    "name": "git"
                }
            ]
        }
    ]'
spec:
  containers:
  - name: run
    image: docker.io/centos/httpd
    ports:
      - containerPort: 80
    volumeMounts:
    - mountPath: /var/www/html
      name: git
  volumes:
  - emptyDir: {}
    name: git
```

这个例子就是用来再应用程序启动前首先从GitHub中拉取代码并存储到共享目录下。

关于Init容器的更详细说明请参考 init容器。

5. 使容器内时间与宿主机同步

我们下载的很多容器内的时区都是格林尼治时间，与北京时间差8小时，这将导致容器内的日志和文件创建时间与实际时区不符，有两种方式解决这个问题：

- 修改镜像中的时区配置文件
- 将宿主机的时区配置文件 `/etc/localtime` 使用volume方式挂载到容器中

第二种方式比较简单，不需要重做镜像，只要在应用的yaml文件中增加如下配置：

```yaml
  volumeMounts:
    - name: host-time
      mountPath: /etc/localtime
      readOnly: true
    volumes:
    - name: host-time
      hostPath:
        path: /etc/localtime
```

6. 在Pod中获取宿主机的主机名、namespace等

这条技巧补充了第一条获取 podIP 的内容，方法都是一样的，只不过列出了更多的引用字段。

参考下面的 pod 定义，每个 pod 里都有一个 {.spec.nodeName} 字段，通过 `fieldRef` 和环境变量，就可以在Pod中获取宿主机的主机名（访问环境变量 `MY_NODE_NAME` ）。

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: dapi-test-pod
spec:
  containers:
    - name: test-container
      image: busybox
      command: [ "/bin/sh", "-c", "env" ]
      env:
        - name: MY_NODE_NAME
          valueFrom:
            fieldRef:
              fieldPath: spec.nodeName
        - name: MY_POD_NAME
          valueFrom:
            fieldRef:
              fieldPath: metadata.name
        - name: MY_POD_NAMESPACE
          valueFrom:
            fieldRef:
              fieldPath: metadata.namespace
        - name: MY_POD_IP
          valueFrom:
            fieldRef:
              fieldPath: status.podIP
        - name: HOST_IP
          valueFrom:
           fieldRef:
             fieldPath: status.hostIP
        - name: MY_POD_SERVICE_ACCOUNT
          valueFrom:
            fieldRef:
              fieldPath: spec.serviceAccountName
  restartPolicy: Never
```

7. 配置Pod使用外部DNS

修改kube-dns的使用的ConfigMap。

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: kube-dns
  namespace: kube-system
data:
  stubDomains: |
    {"k8s.com": ["192.168.10.10"]}
  upstreamNameservers: |
    ["8.8.8.8", "8.8.4.4"]
```

`upstreamNameservers` 即使用的外部DNS，参考：Configuring Private DNS Zones and Upstream Nameservers in Kubernetes

8. 创建一个CentOS测试容器

有时我们可能需要在Kubernetes集群中创建一个容器来测试集群的状态或对其它容器进行操作，这时候我们需要一个操作节点，可以使用一个普通的CentOS容器来实现。yaml文件见manifests/test/centos.yaml。

```yaml
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: test
  labels:
    app: test
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: test
    spec:
      containers:
      - image: harbor-001.jimmysong.io/library/centos:7.2.1511
        name: test
        command: ["/bin/bash","-c","while true; do sleep 1000; d
one"]
        imagePullPolicy: IfNotPresent
```

即使用一个 `while` 循环保证容器启动时拥有一个前台进程。

也可以直接使用 `kubectl run` 的方式来创建：

```
kubectl run --image=harbor-001.jimmysong.io/library/centos:7.2.1511 --command '/bin/bash -c "while true;do sleep 1000;done"' centos-test
```

9. 强制删除一直处于Terminating状态的Pod

有时候当我们直接删除Deployment/DaemonSets/StatefulSet等最高级别的Kubernetes资源对象时，会发现有些改对象管理的Pod一直处于Terminating而没有被删除的情况，这时候我们可以使用如下方式来强制删除它：

一、使用**kubectl**中的强制删除命令

```
kubectl delete pod $POD_ID --force --grace-period=0
```

如果这种方式有效，那么恭喜你！如果仍然无效的话，请尝试下面第二种方法。

二、直接删除**etcd**中的数据

> 这是一种最暴力的方式，我们不建议直接操作etcd中的数据，在操作前请确认知道你是在做什么。

假如要删除 `default` namespace下的pod名为 `pod-to-be-deleted-0` ，在etcd所在的节点上执行下面的命令，删除etcd中保存的该pod的元数据：

```
ETCDCTL_API=3 etcdctl del /registry/pods/default/pod-to-be-deleted-0
```

这时API server就不会再看到该pod的信息。

如何使用etcdctl查看etcd中包括的kubernetes元数据，请参考：使用etcdctl访问kubernetes数据

问题记录

安装、使用kubernetes的过程中遇到的所有问题的记录。

推荐直接在Kubernetes的GitHub上提issue，在此记录所提交的issue。

1.Failed to start ContainerManager failed to initialise top level QOS containers #43856

重启kubelet时报错，目前的解决方法是：

1.在docker.service配置中增加的 `--exec-opt native.cgroupdriver=systemd` 配置。

2.手动删除slice（貌似不管用）

3.重启主机，这招最管用😄

```
for i in $(systemctl list-unit-files —no-legend —no-pager -l | grep —color=never -o .*.slice | grep kubepod);do systemctl stop $i;done
```

上面的几种方法在该bug修复前只有重启主机管用，该bug已于2017年4月27日修复，merge到了master分支，见 https://github.com/kubernetes/kubernetes/pull/44940

2.High Availability of Kube-apiserver #19816

API server的HA如何实现？或者说这个master节点上的服务 `api-server` 、 `scheduler` 、 `controller` 如何实现HA？目前的解决方案是什么？

目前的解决方案是api-server是无状态的可以启动多个，然后在前端再加一个nginx或者ha-proxy。而scheduler和controller都是直接用容器的方式启动的。

3.Kubelet启动时Failed to start ContainerManager systemd version does not support ability to start a slice as transient unit

CentOS系统版本7.2.1511

kubelet启动时报错systemd版本不支持start a slice as transient unit。

尝试升级CentOS版本到7.3，看看是否可以修复该问题。

与kubeadm init waiting for the control plane to become ready on CentOS 7.2 with kubeadm 1.6.1 #228类似。

另外有一个使用systemd管理kubelet的proposal。

4.kube-proxy报错kube-proxy[2241]: E0502 15:55:13.889842 2241 conntrack.go:42] conntrack returned error: error looking for path of conntrack: exec: "conntrack": executable file not found in $PATH

导致的现象

kubedns启动成功，运行正常，但是service之间无法解析，kubernetes中的DNS解析异常

解决方法

CentOS中安装 `conntrack-tools` 包后重启kubernetes集群即可。

5. Pod stucks in terminating if it has a privileged container but has been scheduled to a node which doesn't allow privilege issue#42568

当pod被调度到无法权限不足的node上时，pod一直处于pending状态，且无法删除pod，删除时一直处于terminating状态。

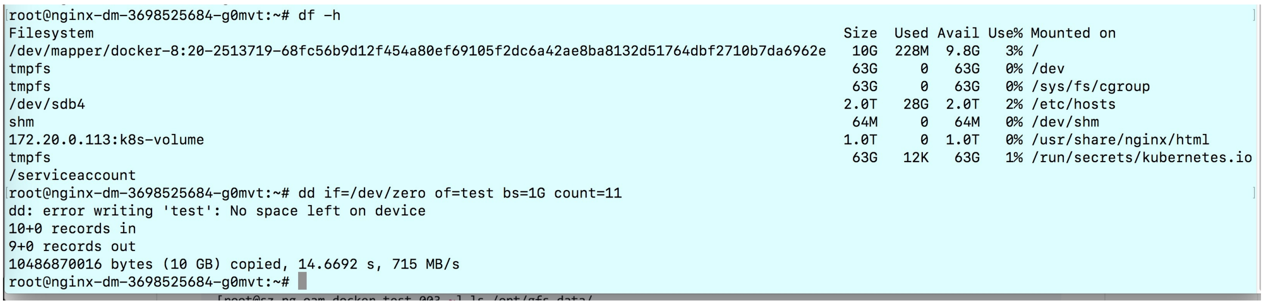
kubelet中的报错信息

```
Error validating pod kube-keepalived-vip-1p62d_default(5d79ccc0-3173-11e7-bfbd-8af1e3a7c5bd) from api, ignoring: spec.containers[0].securityContext.privileged: Forbidden: disallowed by cluster policy
```

6.PVC中对Storage的容量设置不生效

使用glusterfs做持久化存储文档中我们构建了PV和PVC，当时给 `glusterfs-nginx` 的PVC设置了8G的存储限额， `nginx-dm` 这个Deployment使用了该PVC，进入该Deployment中的Pod执行测试：

```
dd if=/dev/zero of=test bs=1G count=10
```



图片 - *pvc-storage-limit*

从截图中可以看到创建了9个size为1G的block后无法继续创建了，已经超出了8G的限额。

7. 使用 Headless service 的时候 kubedns 解析不生效

kubelet 的配置文件 `/etc/kubernetes/kubelet` 中的配置中将集群 DNS 的 domain name 配置成了 `--cluster-domain=cluster.local.` ，虽然对于 service 的名字能够正常的完成 DNS 解析，但是对于 headless service 中的 pod 名字解析不了，查看 pod 的 `/etc/resolv.conf` 文件可以看到以下内容：

```
nameserver 10.0.254.2
search default.svc.cluster.local. svc.cluster.local. cluster.loc
al. jimmysong.io
options ndots:5
```

修改 `/etc/kubernetes/kubelet` 文件中的 `--cluster-domain=cluster.local.` 将 local 后面的点去掉后重启所有的 kubelet，这样新创建的 pod 中的 `/etc/resolv.conf` 文件的 DNS 配置和解析就正常了。

8. kubernetes 集成 ceph 存储 rbd 命令组装问题

kubernetes 使用 ceph 创建 PVC 的时候会有如下报错信息：

```
Events:
  FirstSeen     LastSeen     Count     From                    SubObjec
tPath     Type         Reason              Message
  ---------     --------     -----     ----                    --------
-----     --------     ------              -------
  1h         12s         441    {persistentvolume-controller }
        Warning         ProvisioningFailed     Failed to provision
 volume with StorageClass "ceph-web": failed to create rbd image
: executable file not found in $PATH, command output:
```

检查 `kube-controller-manager` 的日志将看到如下错误信息：

```
Sep  4 15:25:36 bj-xg-oam-kubernetes-001 kube-controller-manager
: W0904 15:25:36.032128   13211 rbd_util.go:364] failed to creat
e rbd image, output
Sep  4 15:25:36 bj-xg-oam-kubernetes-001 kube-controller-manager
: W0904 15:25:36.032201   13211 rbd_util.go:364] failed to creat
e rbd image, output
Sep  4 15:25:36 bj-xg-oam-kubernetes-001 kube-controller-manager
: W0904 15:25:36.032252   13211 rbd_util.go:364] failed to creat
e rbd image, output
Sep  4 15:25:36 bj-xg-oam-kubernetes-001 kube-controller-manager
: E0904 15:25:36.032276   13211 rbd.go:317] rbd: create volume f
ailed, err: failed to create rbd image: fork/exec /usr/bin/rbd:
invalid argument, command output:
```

该问题尚未解决，参考 Error creating rbd image: executable file not found in $PATH#38923

9. Helm: Error: no available release name found

在开启了RBAC的kubernetes集群中，当使用helm部署应用，执行 `helm install` 的时候，会报着个错误：

```
Error: no available release name found
Error: the server does not allow access to the requested resourc
e (get configmaps)
```

这是因为我们使用的 `2.3.1` 版本的helm init的时候没有为tiller创建 `serviceaccount` 和 `clusterrolebiding` 的缘故导致的。

```
kubectl create serviceaccount --namespace kube-system tiller
kubectl create clusterrolebinding tiller-cluster-rule --clusterr
ole=cluster-admin --serviceaccount=kube-system:tiller
# helm init -i harbor-001.jimmysong.io/library/kubernetes-helm-t
iller:v2.3.1
kubectl patch deploy --namespace kube-system tiller-deploy -p '{
"spec":{"template":{"spec":{"serviceAccount":"tiller"}}}}'
```

参考

- Helm: Error: no available release name found - StackOverflow
- Helm 2.2.3 not working properly with kubeadm 1.6.1 default RBAC rules #2224

参考

Persistent Volume

Resource Design Proposals

Helm: Error: no available release name found

Kubernetes版本更新日志

刚开始写作本书时kubernetes1.6刚刚发布，随后kubernetes基本以每3个月发布一个版本的速度不断迭代，为了追踪不同版本的新特性，我们有必要在此记录一下。

每个kubernetes版本的详细更新日志请参考：https://github.com/kubernetes/kubernetes/blob/master/CHANGELOG.md

发布记录

- 2017年6月29日 Kubernetes1.7发布
- 2017年9月28日 Kubernetes1.8发布
- 2017年12月15日 Kubernetes1.9发布

更多

追踪 Kubernetes 最新特性，请访问互动教学网站提供商 Katacoda.com 创建的 kubernetesstatus.com。

Kubernetes1.7更新日志

2017年6月29日，kuberentes1.7发布。该版本的kubernetes在安全性、存储和可扩展性方面有了很大的提升。改版本的详细更新文档请查看Changelog。

这些新特性中包含了安全性更高的加密的secret、pod间通讯的网络策略，限制kubelet访问的节点授权程序以及客户端/服务器TLS证书轮换。

对于那些在Kubernetes上运行横向扩展数据库的人来说，这个版本有一个主要的特性，可以为StatefulSet添加自动更新并增强DaemonSet的更新。我们还宣布了对本地存储的Alpha支持，以及用于更快地缩放StatefulSets的突发模式。

此外，对于高级用户，此发行版中的API聚合允许使用用于自定义的API与API server同时运行。其他亮点包括支持可扩展的准入控制器，可插拔云供应商程序和容器运行时接口（CRI）增强功能。

新功能

安全

- Network Policy API 提升为稳定版本。用户可以通过使用网络插件实现的网络策略来控制哪些Pod之间能够互相通信。
- 节点授权和准入控制插件是新增加的功能，可以用于限制kubelet可以访问的secret、pod和其它基于节点的对象。
- 加密的Secret和etcd中的其它资源，现在是alpha版本。
- Kubelet TLS bootstrapping现在支持客户端和服务器端的证书轮换。
- 由API server存储的审计日志现在更具可定制性和可扩展性，支持事件过滤和webhook。它们还为系统审计提供更丰富的数据。

有状态负载

- StatefulSet更新是1.7版本的beta功能，它允许使用包括滚动更新在内的一系列更新策略自动更新诸如Kafka，Zookeeper和etcd等有状态应用程序。
- StatefulSets现在还支持对不需要通过Pod管理策略进行排序的应用程序进行快速扩展和启动。这可以是主要的性能改进。
- 本地存储（alpha）是有状态应用程序最常用的功能之一。用户现在可以通过标准的PVC/PV接口和StatefulSet中的StorageClass访问本地存储卷。

- DaemonSet——为每个节点创建一个Pod，现在有了更新功能，在1.7中增加了智能回滚和历史记录功能。
- 新的StorageOS Volume插件可以使用本地或附加节点存储中以提供高可用的集群范围的持久卷。

可扩展性

- 运行时的API聚合是此版本中最强大的扩展功能，允许高级用户将Kubernetes风格的预先构建的第三方或用户创建的API添加到其集群中。
- 容器运行时接口（CRI）已经增强，可以使用新的RPC调用从运行时检索容器度量。 CRI的验证测试已经发布，与containerd进行了Alpha集成，现在支持基本的生命周期和镜像管理。参考深入介绍CRI的文章。

其它功能

- 引入了对外部准入控制器的Alpha支持，提供了两个选项，用于向API server添加自定义业务逻辑，以便在创建对象和验证策略时对其进行修改。
- 基于策略的联合资源布局提供Alpha版本，用于根据自定义需求（如法规、定价或性能）为联合（federated）集群提供布局策略。

弃用

- 第三方资源（TPR）已被自定义资源定义（Custom Resource Definitions，CRD）取代，后者提供了一个更清晰的API，并解决了TPR测试期间引发的问题和案例。如果您使用TPR测试版功能，则建议您迁移，因为它将在Kubernetes 1.8中被移除。

以上是Kubernetes1.7中的主要新特性，详细更新文档请查看Changelog。

参考

Kuberentes 1.7: Security Hardening, Stateful Application Updates and Extensibility

Kubernetes1.8更新日志

2017年9月28日，kubernetes1.8版本发布。该版本中包括了一些功能改进和增强，并增加了项目的成熟度，将强了kubernetes的治理模式，这些都将有利于kubernetes项目的持续发展。

聚焦安全性

Kubernetes1.8的基于角色的访问控制（RBAC）成为stable支持。RBAC允许集群管理员动态定义角色对于Kubernetes API的访问策略。通过网络策略筛选出站流量的Beta支持，增强了对入站流量进行过滤的现有支持。 RBAC和网络策略是强化Kubernetes内组织和监管安全要求的两个强大工具。

Kubelet的传输层安全性（TLS）证书轮换成为beta版。自动证书轮换减轻了集群安全性运维的负担。

聚焦工作负载支持

Kubernetes 1.8通过apps/v1beta2组和版本推动核心工作负载API的beta版本。Beta版本包含当前版本的Deployment、DaemonSet、ReplicaSet和StatefulSet。 工作负载API是将现有工作负载迁移到Kubernetes以及开发基于Kubernetes的云原生应用程序提供了基石。

对于那些考虑在Kubernetes上运行大数据任务的，现在的工作负载API支持运行kubernetes原生支持的Apache Spark。

批量工作负载，比如夜间ETL工作，将从CronJobs的beta版本中受益。

自定义资源定义（CRD）在Kubernetes 1.8中依然为测试版。CRD提供了一个强大的机制来扩展Kubernetes和用户定义的API对象。 CRD的一个用例是通过Operator Pattern自动执行复杂的有状态应用，例如键值存储、数据库和存储引擎。随着稳定性的继续推进，预计将继续加强对CRD的验证。

更多

Volume快照、PV调整大小、自动taint、pod优先级、kubectl插件等！

除了稳定现有的功能，Kubernetes 1.8还提供了许多预览新功能的alpha版本。

社区中的每个特别兴趣小组（SIG）都在继续为所在领域的用户提供更多的功能。有关完整列表，请访问发行说明。

参考

Kubernetes 1.8: Security, Workloads and Feature Depth

Kubernetes1.9更新日志

2017年12月15日，kubernetes1.9版本发布。Kubernetes依然按照每三个月一个大版本发布的速度稳定迭代，这是今年发布的第四个版本，也是今年的最后一个版本，该版本最大的改进是Apps Workloads API成为稳定版本，这消除了很多潜在用户对于该功能稳定性的担忧。还有一个重大更新，就是测试支持了Windows了，这打开了在kubernetes中运行Windows工作负载的大门。

Workloads API GA

apps/v1 Workloads API成为GA（General Availability），且默认启用。 Apps Workloads API将**DaemonSet**、**Deployment**、**ReplicaSet**和**StatefulSet** API组合在一起，作为Kubernetes中长时间运行的无状态和有状态工作负载的基础。

Deployment和ReplicaSet是Kubernetes中最常用的两个对象，经过一年多的实际使用和反馈后，现在已经趋于稳定。SIG apps同时将这些经验应用到另外的两个对象上，使得DaemonSet和StatefulSet也能顺利毕业走向成熟。v1（GA）意味着已经生产可用，并保证长期的向后兼容。

Windows支持（beta）

Kubernetes最初是为Linux系统开发的，但是用户逐渐意识到容器编排的好处，我们看到有人需要在Kubernetes上运行Windows工作负载。在12个月前，我们开始认真考虑在Kubernetes上支持Windows Server的工作。 SIG-Windows现在已经将这个功能推广到beta版本，这意味着我们可以评估它的使用情况。

增强存储

kubernetes从第一个版本开始就支持多种持久化数据存储，包括常用的NFS或iSCSI，以及对主要公共云和私有云提供商的存储解决方案的原生支持。随着项目和生态系统的发展，Kubernetes的存储选择越来越多。然而，为新的存储系统添加volume插件一直是一个挑战。

容器存储接口（**CSI**）是一个跨行业标准计划，旨在降低云原生存储开发的障碍并确保兼容性。 SIG-Storage和CSI社区正在合作提供一个单一接口，用于配置、附着和挂载与Kubernetes兼容的存储。

Kubernetes 1.9引入了容器存储接口（CSI）的alpha实现，这将使挂载新的volume插件就像部署一个pod一样简单，并且第三方存储提供商在开发他们的解决方案时也无需修改kubernetes的核心代码。

由于该功能在1.9版本中为alpha，因此必须明确启用该功能，不建议用于生产使用，但它为更具扩展性和基于标准的Kubernetes存储生态系统提供了清晰的路线图。

其它功能

自定义资源定义（CRD）校验，现在已经成为beta，默认情况下已启用，可以用来帮助CRD作者对于无效对象定义给出清晰和即时的反馈。

SIG Node硬件加速器转向alpha，启用GPU，从而实现机器学习和其他高性能工作负载。

CoreDNS alpha可以使用标准工具来安装CoreDNS。

kube-proxy的IPVS模式进入beta版，为大型集群提供更好的可扩展性和性能。

社区中的每个特别兴趣小组（SIG）继续提供其所在领域的用户最迫切需要的功能。有关完整列表，请访问发行说明。

获取

Kubernetes1.9已经可以通过GitHub下载。

参考

Kubernetes 1.9: Apps Workloads GA and Expanded Ecosystem

Kubernetes1.10更新日志

2018年3月26日，kubernetes1.10版本发布，这是2018年发布的第一个版本。该版本的Kubernetes主要提升了Kubernetes的成熟度、可扩展性与可插入性。

该版本提升了三大关键性功能的稳定度，分别为存储、安全与网络。另外，此次新版本还引入了外部kubectl凭证提供程序（处于alpha测试阶段）、在安装时将默认的DNS服务切换为CoreDNS（beta测试阶段）以及容器存储接口（简称CSI）与持久化本地卷的beta测试版。

下面再分别说下三大关键更新。

存储

- CSI（容器存储接口）迎来Beta版本，可以通过插件的形式安装存储。
- 持久化本地存储管理也迎来Beta版本。
- 对PV的一系列更新，可以自动阻止Pod正在使用的PVC的删除，阻止已绑定到PVC的PV的删除操作，这样可以保证所有存储对象可以按照正确的顺序被删除。

安全

- kubectl可以对接不同的凭证提供程序
- 各云服务供应商、厂商以及其他平台开发者现在能够发布二进制插件以处理特定云供应商IAM服务的身价验证

网络

- 将原来的kube-dns切换为CoreDNS

获取

Kubernetes1.10已经可以通过GitHub下载。

参考

[Kubernetes 1.10: Stabilizing Storage, Security, and Networking]

Kubernetes及云原生年度总结及展望

本节将聚焦Kubernetes及云原生技术的年度总结并展望下一年的发展。

Kubernetes与云原生2017年年终总结及2018年展望

本文主要关于Kubernetes及云原生生态圈在2017年取得的进展，及对2018年的展望。

云计算技术发展至今已经10多个年头了，从最开始的硬件虚拟化、IaaS、OpenStack、PaaS、容器设置到Serverless发展至今，已经越来约接近应用逻辑，容器实现了应用的分装，方便了应用在不同环境间的迁移，轻量级的特性又使它能够消耗更少的资源而带来更多的便利，但是独木难支，容器如果在单节点上运行并不能发挥它的最大效益，容器编排领域在2016年就成为了兵家必争之地。在新的一年即将到来时，本文将带您一起梳理2017年Kubernetes及云原生的发展，并对其在2018年的趋势作出预测。

Kubernetes

谈到Kuberentes就不得不谈到容器，容器从几年前的大热到现在的归于平淡，之前大家说容器通常是指Docker容器，甚至很多人就将容器等同于Docker，还有很多人像操作虚拟机一样得使用容器。

Kubernetes是谷歌根据其内部使用的Borg改造成一个通用的容器编排调度器，于2014年将其发布到开源社区，并于2015年将其捐赠给Linux基金会的下属的云原生计算基金会（CNCF），也是GIFEE（Google Infrastructure For Everyone Else）中的一员，其他还包括HDFS、Hbase、Zookeeper等，见https://github.com/GIFEE/GIFEE，下面就让我们来回顾一下Kubernetes的技术发展史。

Kubernetes发展历史

相信凡是关注容器生态圈的人都不会否认，Kubernetes已经成为容器编排调度的实际标准，不论Docker官方还是Mesos都已经支持Kubernetes，Docker公司在今年10月16日至19日举办的DockerCon EU 2017大会上宣布支持Kubernetes调度，就在这不久前Mesos的商业化公司Mesosphere的CTO Tobi Knaup也在官方博客中宣布Kubernetes on DC/OS。而回想下2016年时，我们还在为Swarm、Mesos、

Kubernetes谁能够在容器编排调度大战中胜出而猜测时，而经过不到一年的发展，Kubernetes就以超过70%的市场占有率（据TheNewStack的调研报告）将另外两者遥遥的甩在了身后，其已经在大量的企业中落地，还有一些重量级的客户也宣布将服务迁移到Kubernetes上，比如GitHub（见Kubernetes at GitHub），还有eBay、彭博社等。

Kubernetes自2014年由Google开源以来，至今已经发展到了1.9版本，下面是Kubernetes的版本发布路线图：

- 2014年10月由Google正式开源。
- 2015年7月22日发布1.0版本，在OSCON（开源大会）上发布了1.0版本。
- 2015年11月16日发布1.1版本，性能提升，改进了工具并创建了日益强大的社区。
- 2016年4月16日发布1.2版本，更多的性能升级加上简化应用程序部署和管理。
- 2016年7月22日发布1.3版本，对接了云原生和企业级工作负载。
- 2016年9月26日发布1.4版本，该版本起Kubernetes开始支持不同的运行环境，并使部署变得更容易。
- 2016年12月13日发布1.5版本，该版本开始支持生产级别工作负载。
- 2017年3月28日发布1.6版本，该版本支持多租户和在集群中自动化部署不同的负载。
- 2017年6月29日发布1.7版本，该版本的kubernetes在安全性、存储和可扩展性方面有了很大的提升。
- 2017年9月28日发布1.8版本，该版本中包括了一些功能改进和增强，并增加了项目的成熟度，将强了kubernetes的治理模式，这些都将有利于kubernetes项目的持续发展。
- 2017年12月15日发布1.9版本，该版本最大的改进是Apps Workloads API成为稳定版本，这消除了很多潜在用户对于该功能稳定性的担忧。还有一个重大更新，就是测试支持了Windows了，这打开了在kubernetes中运行Windows工作负载的大门。

从上面的时间线中我们可以看到，Kubernetes的产品迭代周期越来越快，从2014年开源，2015年发布了两个版本，2016年发布了三个版本，而今年一年内就发布了4个大版本，Kubernetes已经变了的越来越稳定，越来越易用。

Kubernetes的架构做的足够开放，通过系列的接口，如CRI（Container Runtime Interface）作为Kubelet与容器之间的通信接口、CNI（Container Networking Interface)来管理网络、而持久化存储通过各种Volume Plugin来实现，同时

Kubernetes的API本身也可以通过CRD（Custom Resource Define）来扩展，还可以自己编写Operator和Service Catalog来基于Kubernetes实现更高级和复杂的功能。

Cloud Native

在Kubernetes出现之前，就已经有人提出了云原生的概念，如2010年Paul Fremantle就在他的博客中提出了云原生的核心理念，但是还没有切实的技术解决方案。而那时候PaaS才刚刚出现，PaaS平台提供商Heroku提出了12因素应用的理念，为构建SaaS应用提供了方法论，该理念在云原生时代依然适用。

现如今云已经可以为我们提供稳定的可以唾手可得的基础设施，但是业务上云成了一个难题，Kubernetes的出现与其说是从最初的容器编排解决方案，倒不如说是为了解决应用上云（即云原生应用）这个难题。CNCF中的托管的一系列项目即致力于云原生应用整个生命周期的管理，从部署平台、日志收集、Service Mesh（服务网格）、服务发现、分布式追踪、监控以及安全等各个领域通过开源的软件为我们提供一揽子解决方案。

国外已经有众多的Kubernetes和Cloud Native meetup定期举办，在中国今年可以说是小荷才露尖尖角。

- 2017年6月19日-20日，北京，L3大会（LinuxCon+ContainerCon+CloudOpen China）。CNCF（Cloud Native Computing Foundation）作为云原生应用的联合推广团体，也是由Google一手培植起来的强大“市场媒体”（Kubernetes是第一个入选该基金会的项目），第一次进入中国，华为、Google、Rancher、红帽等公司分别做了关于Kubernetes及Cloud Native的演讲。
- 2017年7月25日，北京、上海，k8smeetup，Kubernetes二周年北京-上海Meetup双城庆生。
- 2017年9月12日，北京，T11大会，前Pivotal技术专家，现CapitalOne高级专家Kevin Hoffman做了High Level Cloud Native Concepts的演讲。
- 2017年10月15日，杭州，KEUC 2017- Kubernetes 中国用户大会。由才云科技（Caicloud）、美国 The Linux Foundation 基金会旗下 Cloud Native Computing Foundation (CNCF)、「K8sMeetup 中国社区」联合主办的聚焦Kubernetes中国行业应用与技术落地的盛会。
- 2017年12月13日-15日，杭州，云原生技术大会——CNTC。这次会议由谐云科技与网易云共同主办，主要探讨云原生技术与应用，同时还进行了云原生集训。

另外还有由才云科技分别在北京、上海、深圳、青岛等地举办了多场k8smeetup。

容器是云原生的基石

容器最初是通过开发者工具而流行，可以使用它来做隔离的开发测试环境和持续集成环境，这些都是因为容器轻量级，易于配置和使用带来的优势，docker和docker-compose这样的工具极大的方便的了应用开发环境的搭建，同时基于容器的CI/CD工具如雨后春笋般出现。

隔离的环境、良好的可移植性、模块化的组件、易于扩展和轻量级的特性，使得容器成为云原生的基石。但是容器不光是docker一种，还有cri-o、rkt等支持OCI标准的容器，以及OpenStack基金会推出的兼容容器标准的号称是轻量级虚拟机的Kata Containers，Kubernetes并不绑定到某一容器引擎，而是支持所有满足OCI运行时标准的容器。

下一代云计算标准

Google通过将云应用进行抽象简化出的Kubernetes中的各种概念对象，如Pod、Deployment、Job、StatefulSet等，形成了Cloud Native应用的通用的可移植的模型，Kubernetes作为云应用的部署标准，直接面向业务应用，将大大提高云应用的可移植性，解决云厂商锁定的问题，让云应用可以在跨云之间无缝迁移，甚至用来管理混合云，成为企业IT云平台的新标准。

现状及影响

Kubernetes既然是下一代云计算的标准，那么它当前的现状如何，距离全面落地还有存在什么问题？

当前存在的问题

如果Kubernetes被企业大量采用，将会是对企业IT价值的重塑，IT将是影响业务速度和健壮性的中流砥柱，但是对于Kubernetes真正落地还存在诸多问题：

- 部署和运维起来复杂，需要有经过专业的培训才能掌握；
- 企业的组织架构需要面向DevOps转型，很多问题不是技术上的，而是管理和心态上的；

- 对于服务级别尤其是微服务的治理不足，暂没有一套切实可行可落地的完整微服务治理方案；
- 对于上层应用的支持不够完善，需要编写配置大量的YAML文件，难于管理；
- 当前很多传统应用可能不适合迁移到Kuberentes，或者是成本太高，因此可以落地的项目不多影响推广；

以上这些问题是企业真正落地Kubernetes时将会遇到的比较棘手的问题，针对这些问题，Kubernetes社区早就心领神会有多个SIG)（Special Interest Group）专门负责不同领域的问题，而初创公司和云厂商们也在虎视眈眈觊觎这份大蛋糕。

日益强大的社区

Kubernetes已成为GitHub上参与和讨论人数最多的开源项目，在其官方Slack上有超过两万多名注册用户（其中包括中文用户频道**cn-users**），而整个Kubernetes中文用户群可达数千名之众。

目前关于Kubernetes和云原生图书也已经琳琳总总，让人眼花缭乱。

英文版的讲解Kubernetes的书籍有：The Kubernetes Book、Kubernetes in Action、Kubernetes Microservices with Docker，关于云原生架构的Cloud Native Infrastructure: Patterns for Scalable Infrastructure and Applications in a Dynamic Environment等已发行和2018年即将发行的有十几本之多，同时还有关于云原生开发的书籍也鳞次栉比，如Cloud Native Go（这本书已经被翻译成中文，由电子工业出版社引进出版）、Cloud Native Python（已由电子工业出版社引进，预计2018年推出中文版），Cloud Native Java等。

关于Kuberentes和云原生的中文版的书籍有：《Kubernetes权威指南:从Docker到Kubernetes实践全接触》，《Java云原生》（预计2018年出版），还有一系列开源的电子书和教程，比如我写的kubernetes-handbook，同时Kubernetes官方官网文档也即将推出完整的汉化版本，该项目目前还在进行中，见kubernetes-docs-cn。

另外，除了图书和官方Slack外，在中国还有很多厂商、社区、爱好者组织的meetup、微信群推广Kubernetes，同时吸引了大量的用户关注和使用Kubernetes。

创业公司与厂商支持

国外的Google的GKE、微软的Azure ACS、AWS的Fargate和2018年即将推出的EKS、Rancher联合Ubuntu推出的RKE，国内的华为云、腾讯云、阿里云等都已推出了公有云上的Kuberentes服务，Kubernetes已经成为公有云的容器部署的标配，私有云领域也有众多厂商在做基于Kubernetes的PaaS平台。随着企业落地Kubernetes的日益增长，相关的人才缺口也将日益显现。CNCF又就此推出了CKA（Certified Kubernetes Administrator）和CKD（Certified Kubernetes Developer），假若在Kubernetes的生态构建与市场发展顺利的情况下，该证书将会展现其含金量。

另外在国外还有一大批基于Kubernetes的创业公司，如Kubernetes创始人之一Joe Beda创立了Heptio（于今年9月获获得2500万美元B轮融资），还有Platform9、Kismatic、Diamanti、Bitnami、CoreOS、Hypernetes、Weave、NavOps等，他们中有的提供Kubernetes的技术咨询和培训，有的专研某项具体技术，还有一系列基于Kubernetes的自动化工具、监控厂商如雨后春笋般出现。

国内前几年诞生了多家容器创业公司，例如DaoCloud、精灵云、时速云、数人云、灵雀云、有容云、好雨云、希云、才云、博云等，这些厂商有的可能一开始不是基于Kubernetes作为容器编排调度引擎，但是现在已经全部支持，其中灵雀云于11月8日获得腾讯云领投的B轮融资。这些容器厂商全部涉及私有云业务，主要对接金融、政府和电信行业，帮助传统企业进行IT转型，虽然很多厂商都生成支持Kubernetes，但是在Kubernetes的易用性上还需要很多改进，单纯基于容器部署应用已经无法满足企业的需求，帮助企业上云、将传统应用改造以适应云的弹性与高效，构建PaaS平台，通过基于容器的基础调度平台运行大数据及AI应用，成为创业公司的众矢之的，对于特定行业的整体的解决方案将是国内的容器厂商的主要商业化方式。

目前大部分容器云提供的产品大同小异，从云平台管理、容器应用的生命周期管理、DevOps、微服务架构等，这些大多是对原有应用的部署和资源申请流程的优化，没有形成杀手级的平台级服务，这些都是原来容器时代的产物。而容器云进化到高级阶段Cloud Native（云原生）后，容器技术将成为该平台的基础，虽然大家都生成具有全面的功能，但是厂商在推行容器技术时需要结合企业的具体应用场景下进行优化。

2018年展望

2017年可以说是Cloud Native蓬勃发展和大发异彩之年，Kuberentes在这一年中连续发布了4个版本，从1.6到1.9，Containerd、Fluentd、CoreDNS、Jeager分别发布自己的1.0版本。

在今年12月的KubeCon&CloudNativeCon Austin会议上，已经为2018年的云原生生态圈的发展确定几大关键词：

- 服务网格（Service Mesh），在Kubernetes上践行微服务架构进行服务治理所必须的组件；
- 无服务器架构（Serverless），以FaaS为代表的无服务器架构将会流行开来；
- 加强数据服务承载能力，例如在Kubernetes上运行大数据应用；
- 简化应用部署与运维包括云应用的监控与日志收集分析等；

这些功能是Kubernetes生态已有但是亟待加强的功能，它们能够解决我们在上文中提到的当前生态中存在的问题。

2018年的IaaS的运营商将主要提供基础架构服务，如虚拟机、存储和数据库等传统的基础架构和服务，仍然会使用现有的工具如Chef、Terraform、Ansible等来管理；Kubernetes则可能直接运行在裸机上运行，结合CI/CD成为DevOps的得力工具，并成为高级开发人员的应用部署首选；Kubernetes也将成为PaaS层的重要组成部分，为开发者提供应用程序部署的简单方法，但是开发者可能不会直接与Kubernetes或者PaaS交互，实际的应用部署流程很可能落在自动化CI工具如Jenkins上。

2018年，Kubernetes将更加稳定好用，云原生将会出现更多的落地与最佳实践，这都值得我们期待！

Kubernetes认证服务提供商（KCSP）

云原生计算基金会（CNCF）负责维护并整合Kubernetes和Prometheus之类的开源技术，今天它在开源峰会上宣布了22多家Kubernetes认证服务提供商（KCSP）的创始成员名单。KCSP是通过初审的企业组织，它们在帮助企业成功地采用Kubernetes方面有着丰富经验。此外，专业人员个人现在可以注册报名新的认证Kubernetes管理员（CKA）计划和考试。 KCSP的创始成员包括CNCF和Linux基金会的下列成员：埃森哲、博思艾伦、Bitnami、Canonical、Container Solutions、CoreOS、Ghostcloud、Giant Swarm、Heptio、华为、IBM、inwinSTACKInc.、LiveWyer、Mirantis、RX- M LLC、三星SDS、Stackpointcloud、Supergiant和Treasure Data。 KCSP计划的适用对象是通过初审的服务提供商，它们为踏上Kubernetes之旅的公司企业提供Kubernetes支持、咨询、专业服务和培训。KCSP计划确保企业得到它们在寻求的支持，从而比以前更迅速、更高效地部署新的应用程序，同时确保有一家值得信赖、经过审查的合作伙伴可以支持其生产和运营方面的要求。 想成为KCSP，需要至少三名工程师通过认证**Kubernetes**管理员 CKA 考试，能够证明在Kubernetes社区从事活动（包括积极贡献代码），以及支持企业最终用户的商业模式，包括将工程师派驻客户现场。

> 参考 CNCF 宣布首批 Kubernetes 认证服务提供商

认证Kubernetes管理员（CKA）

这是CNCF的官方认证,详情请看官方介绍

认证详情：

在3~4小时内用命令行进行排障， 解决问题， 相关知识点和权重

- Installation, Configuration & Validation 安装，配置和验证12%
    - 设计一个k8s 集群
    - 安装k8s master 和 nodes
    - 配置安全的集群通信
    - 配置高可用的k8s集群
    - 知道如何获取k8s的发行的二进制文件

  - 提供底层的基础措施来部署一个集群
  - 选择一个网络方案
  - 选择你的基础设施配置
  - 在你的集群上配置端对端的测试
  - 分析端对端测试结果
  - 运行节点的端对端测试
- Core Concepts 核心概念 19%
  - 理解k8s api原语
  - 理解k8s 架构
  - 理解services和其它网络相关原语
- Application Lifecycle Management 应用生命周期管理 8%
  - 理解Deployment， 并知道如何进行rolling update 和 rollback
  - 知道各种配置应用的方式
  - 知道如何为应用扩容
  - 理解基本的应用自愈相关的内容
- Networking 网络 11%
  - 理解在集群节点上配置网络
  - 理解pod的网络概念
  - 理解service networking
  - 部署和配置网络负载均衡器
  - 知道如何使用ingress 规则
  - 知道如何使用和配置cluster dns
  - 理解CNI
- Storage 存储 7%
  - 理解持久化卷（pv），并知道如何创建它们
  - 理解卷（volumes）的access mode
  - 理解持久化卷声明（pvc）的原语
  - 理解k8s的存储对象（kubernetes storage objects）
  - 知道如何为应用配置持久化存储
- Scheduling 调度 5%
  - 使用label选择器来调度pods
  - 理解Daemonset的角色
  - 理解resource limit 会如何影响pod 调度
  - 理解如何运行多个调度器， 以及如何配置pod使用它们
  - 不使用调度器， 手动调度一个pod
  - 查看和显示调度事件events

    - 知道如何配置kubernetes scheduler
- Security 安全 12%
    - 知道如何配置认证和授权
    - 理解k8s安全相关原语
    - 理解如何配置网络策略（network policies）
    - 配合使用镜像的安全性
    - 定义安全上下文
    - 安全的持久化保存键值
- Cluster Maintenance 集群维护 11%
    - 理解k8s的集群升级过程
    - 促进操作系统的升级
    - 补充备份和还原的方法论
- Logging / Monitoring 日志/监控 5%
    - 理解如何监控所有的集群组件
    - 理解如何监控应用
    - 管理集群组件日志
    - 管理应用日志
- Troubleshooting 问题排查 10%
    - 排查应用失败故障
    - 排查控制层（control panel）故障
    - 排查工作节点（work node）故障
    - 排查网络故障

> 参考课程大纲

考试说明和checklist

- 注册考试
- 检查系统要求
- 选择考试日期
- 获取考生手册
- 验证姓名
- 阅读重要提示
- 参加考试

考试期间，除了考试系统界面上的内容和按钮能操作外，其它的最好不要动。所有考试无关的内容和资料不允许出现，包括手机，参考书等等。

考试的时候会提供一个Linux Server Terminal ，是基于Gateone 的web终端，一些快捷键可能跟一般的linux 终端软件不一样，请提前了解和体验

考试由31个问题组成，需要你用命令来解决这些问题

有8套环境，31个问题在这8套环境里进行操作。

平均每套环境有3~5个问题。

Cluster	Members	CNI	Description
k8s	1 CA, 1 etcd, 1 master, 2 worker	flannel	non-HA k8s cluster
Hk8s	1 CA, 3 etcd, 3 master, 1 loadbalancer, 2 worker	calico	HA k8s cluster
bk8s	1 CA, 1 etcd, 1 master, 1 worker	flannel	non-HA k8s cluster
wk8s	1 CA, 1 etcd, 1 master, 2 worker	flannel	non-HA k8s cluster
ek8s	1 CA, 1 etcd, 1 master, 2 worker	flannel	non-HA k8s cluster
fk8s	1 CA, 1 etcd, 1 base node none k8s cluster	none	missing master node
ik8s	1 CA, 1 etcd, 1 master, 1 base node	flannel	k8s cluster - missing worker node
tk8s	1 CA, 1 etcd, 1 master, 1 worker	flannel	non-HA k8s cluster

图片 - *cka-question*

通过kubectl config use-context 来切换不同的环境（集群使用的的k8s 1.6.2版本）

获取集群信息，要指定对应的context，如 kubectl get no -l name=hk8s-node-1 --context=hk8s

> 参考 官方考试手册

FAQ

1. 考试费用？

> 300美元， 可以重考一次

2. 考试时间长度？

> 最长4个小时，依赖考生的熟练程度

3. 如何监考？

> 屏幕共享软件可以让监考官看到考生的屏幕， 所有的音频， 视频和屏幕共享流会被保留一段时间， 用于审查

4. 系统要求？

> chrome浏览器， 网络连接， 网络摄像头和麦克风 这个连接可以帮忙检查系统要求 ， 注意Select “Linux Foundation” as the Exam Sponsor and “CKA” as the Exam.

5. 考试期间， 我可以利用什么资源么？

> 只能打开考试页面以及Kubernetes官网

6. 考试期间是否可以做笔记？

> 可以做笔记， 但是仅限于在考试控制页面上的工具上

7. 需要什么证件？

> 包含考生照片的官方认证证件， 比如护照， 身份证， 驾驶证等(注意，需要证件上要有你的英文名称的全名， 对中国居民来讲， 可以使用护照) 需要注册一个Linux Foundation的账号， 到这里注册

8. 考试如何打分？

> 24小时内会自动打分， 72~75小时内会邮件发送认证结果

9. 认证的有效期？

> 有效期2年， 在过期之前需要重新考试

10. 取消和重订

> 在预定考试日期前24小时外，取消或重订， 可以获得完整退费

> 参考 CNCF_FAQ

复习资料

Kubernetes-Learning-Resources

Kubernetes-Certified-Administrator

培训课程导图

新手训练营

Linux Foundation 提供的免费入门课程

有个CNCF网站提供了免费的分级课程 和 课程文档

除了认证大纲内容外，还可以看看 feisky的kubernetes指南 里面包含了上述认证的大部分内容

该课程的课程大纲：

```
Welcome & Introduction
Container Orchestration
Kubernetes
Kubernetes Architecture - Overview
Installing Kubernetes
Setting Up a Single Node Kubernetes Cluster Using Minikube
Accessing Minikube
Kubernetes Building Blocks
Services
Deploying a Stand-Alone Application
Kubernetes Volume Management
Deploying a Multi-Tier Application
ConfigMaps and Secrets
Ingress
Advanced Topics - Overview
Kubernetes Community
Final Exam
ps： 个人觉得这个课程可以不用学， 直接看文档就行了 。
```

还有一个收费的课程 ($299)

该课程的课程大纲：

```
Kubernetes Fundamentals
Chapter 1. Course Introduction
Chapter 2. Basics of Kubernetes
Chapter 3. Kubernetes Architecture
Chapter 4. Kubernetes Installation and Configuration
Chapter 5. Accessing a k8s Cluster and Using the API
Chapter 6. Replication Controllers and Deployments
Chapter 7. Volumes and Application Data
Chapter 8. Services
Chapter 9. Ingress
Chapter 10. Additional API Objects
Chapter 11. Scheduling
Chapter 12. Logging, Monitoring, and Troubleshooting
Chapter 13. Third-Party Resources
Chapter 14. Kubernetes Federation
Chapter 15. Helm
Chapter 16. Security
ps： 个人觉得这个课程太贵了，为了省点钱 ， 仔细研究下文档就行了 。
```

培训体系



图片 - *cka-mindmap*

From: Github_hackstoic